PART VI

GETTING SMARTER

Since Part I, you have been able to use objects. However, you have been able to use only readily available objects, because you do not know how to make your own class of objects. In Part VI, you will learn how to create your own classes of objects, and, even smarter, you will learn how to derive your classes from existing classes to *inherit* everything that interests you. Screen Objects will be used again to help you. Finally, your skill in developing applications will be improved.

SKILL

SIXTEEN

MAKING AND MODIFYING CLASSES

- Declaring and defining classes
- Making members public, private, or protected
- Isolating information with encapsulation
- Building constructors
- Using objects
- Creating a new class of objects
- Accessing class members

Objects and classes—you have been using these since Skill 1. It is now time to get to know a little more about them.

You already know that each class of objects knows how to respond to a specific set of messages. For example, `athlete` knows `up`, `ready`, etc. `Clock` knows `reset`, `time`, `wait`, etc. This is because each class of objects has a set of functions that is tied to these objects. They are part (or members) of that class of objects. Figure 16.1 shows an object containing member functions. The program sends *messages* to this object that invoke the object's member functions. The member functions then manipulate the object accordingly.

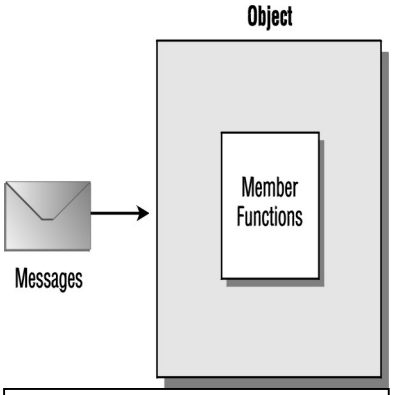

Fig 16.1 Messages to object

These functions are called *member functions* because they belong to and are *members* of the class. Member functions are programmed in the same way that ordinary, *nonmember* functions are programmed. However, as we shall see, member functions are declared to be members of the class, and can only be used with an object of that class.

By now, you are ready to better understand the relationship between a class and an object. *Class* refers to all objects that behave similarly. For example, all circles have a particular shape, a location, a color, and a diameter. *Object* refers to specific instances of that class. For example, Earth has a different size, color, and location than the Moon and the Sun. The relationship between classes and objects is essentially the same as the relationship between types and variables.

Skill 16 will teach you how to make your own classes and how to understand how classes work. Most examples will use Screen Objects, so that you can make classes with interesting applications.

Making Your Own Classes

The athletes you have created and worked with throughout this book have the following member functions, which can be used only with an athlete:

- `ready()`
- `up()`
- `left()`
- `right()`

which means you can do the following:

```
Sal.ready();
```

but not:

```
ready();
```

unless you have a nonmember function named `ready()`, too.

Also, you cannot do the following:

```
myclock.ready();
```

because `Clock` does not have a `ready()` member function.

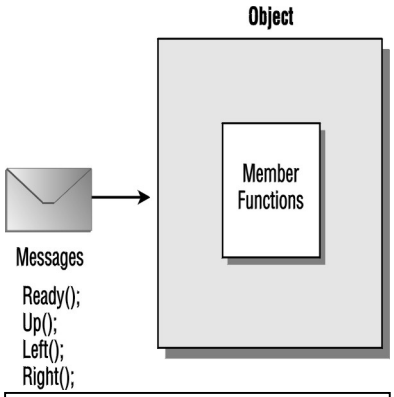

Fig 16.2 Member functions in athlete objects

Since you have been using objects mainly by invoking their member functions, I hope you have a good idea by now of what member functions are. Figure 16.2 shows `athlete` and its member functions.

Another very important aspect of objects may have passed unnoticed until now. Most likely, each object also has some data that determine the status of the object. This is not restricted to our book's objects. Any class can determine that objects belonging to the class can have data as well as functions.

For example, athletes are shown in a position on the screen. Each athlete has an *x* and a *y* coordinate to identify this position. This is done by keeping two variables (for each object) to hold the coordinates. Similarly, the shape of the athlete is determined by a set of rectangles and a circle. Each rectangle is an object of class `Square` (distorted to appear as a rectangle), and the circle is an object of class `Circle`. So, in addition to the variables *x* and *y*, each object also has a `Circle` and five `Squares` as data members, conveniently sized and located.

In another, simpler example, clocks have only one variable as a data member, as we shall see in detail. Since `Clock` objects are supposed to know the elapsed time, there must be a way to know the time when the object was created or reset. This is done by a variable `timestarted` that holds the value of the computer clock (the *hardware* clock) when the object was created or last reset. The member function `time()` simply subtracts `timestarted` from the current time held in the hardware clock to determine the elapsed time. The member function `reset()` just copies the value in the hardware clock to the `timestarted` variable.

Data members can be of any kind. They may be `int`, `float`, `long`, or other common types, and they may also be `athletes`, `Squares`, `Clocks`, or any other available class of objects.

It may be worthwhile to point out that each object of a given class possesses the complete set of member functions and data members defined for that particular class. For example, it is not possible for a circle (such as Earth) to have a color, while another circle (such as the Sun) does not have a color. However, the data members most likely will hold a different value for each object (each athlete has his own coordinates). This means that each object will have its own data. However, unlike the data members, all the member functions are exactly alike, and the same function can be used with every object of that class. This means that only one copy of the member functions for a given class is included in your program.

For example, if you declare two `Clocks`, `clock1` and `clock2`, each one will need to store a different value for `timestarted`, since they were created at different times and may be reset at different times, too. If you declare one thousand objects of class `Clock`, there will be one thousand variables `timestarted`—one for each object—but only one copy of the functions `reset()`, `time()`, and `wait()`.

Building a Class of Objects

You are probably anxious to learn how to build a class of objects. To do this, you will learn how the `Clock` class was built.

Real-life clocks have a dial where the time is shown. All you have to do is to look at the dial to know the time. The `Clock` objects used in our programs work a little differently because of the following:

- They can tell the time elapsed since the clock was created or reset (not since midnight).
- They keep the time in seconds (not in hours, minutes, and seconds).
- They can perform the operations `reset()`, `time()`, `wait()`, and `watch()`.

It is not possible to set our `Clocks` to a given value. They can only be reset.

The clock you have been using in your examples and exercises uses the hardware clock in your computer to keep track of the time.

Building the Clock Code

There is a way to find out the time elapsed since your computer was turned on—it is a function offered by Windows:

```
timeGetTime();
```

If you want to use this function directly, you must include the header file `mmsystem.h`.

This function returns a long integer containing the number of milliseconds elapsed since the system was started. If we keep a variable `timestarted`, we can implement our functions as follows:

```
void reset()
{
    timestarted=timeGetTime();
}
float time()
{
   // Divide result by 1000 because we want
   //      result in seconds, not in 1/1000ths:
   return (timeGetTime()-timestarted)*.001;
}
```

The `wait()` function is somewhat trivial. If we need to wait for a given number of seconds (`tsec`), we can determine the time of the computer clock by adding the current time to `tsec`. We can call this result `willbe`. All we have to do now is to keep looping until the computer clock reaches the same value as `willbe`.

```
void wait(float tsec)
{                          // If we want to wait for tsec
   float willbe;           //      seconds, we have to wait until
   willbe=time()+tsec;     //      the time becomes current time
   while (time()<willbe);//      plus tsec
}
```

Notice that we are using the member function `time()` inside another member function `wait()`. This is perfectly OK. However, you may be confused, because a member function is always supposed to be used in conjunction with an object—here, we are using the `time()` function by itself as if it were a nonmember function.

Implicit References to Objects

It just so happens that the `wait()` function is used with a `Clock` object while the program is running. The `Clock` that was used to `wait()` will be the same clock used with the `time()` function.

Member functions are assumed to operate with the object that was being used.

For example, if you have an object `BigBen` that is declared to be an instance of class `Clock`:

```
Clock BigBen;
```

and you tell `BigBen` to `wait()` in your program:

```
BigBen.wait(10);
```

the `wait()` function automatically assumes that `BigBen` is to be used in conjunction with the `time()` member function when the `wait()` function invokes `time()`.

In fact, it could not work any other way. When you are writing the code for a member function (such as `wait()` and `time()`), you don't really know which object you will have to use. Will it be `mywatch`? Will it be `timer`? Will it be `BigBen`? Therefore, you are not able to use an object name, because you don't know what it is.

All you know is that it will be an object that is an instance of the class `Clock`. It will have a value in the long integer `timestarted`, and it will have the member functions `wait()`, `time()`, and `reset()`. When your member function is invoked, the compiler will remember which object was used, and it will use the data pertaining to that object in all the member function calls that omit the object name.

The implementation of the `wait()` member function restricts the maximum waiting time to 60 seconds. Here is the code:

```
void wait(float tsec)
{
   float willbe;
   if (tsec>60) tsec=60;// Will not wait more than
                        //      1 minute!
   willbe=time()+tsec;
   while (time()<willbe);
}
```

Notice that `while` only compares the value of the current time with the variable `willbe`. It keeps comparing until, eventually, the `time()` function replies with a value that is greater than `willbe`, and then the loop ends. The computer does nothing else in the meantime, because it is busy waiting for the correct time. In a multitasking computer, this is not a good idea.

Declaring a Class

Well, so far we've done nothing new. You already know how to write functions, so you could write the code to `reset()`, `time()`, and `wait()`. We still do not know how to incorporate these functions in a class. Actually, we are very close to implementing classes.

A class is implemented in two steps:

- The *class declaration* gives a name to the class, and explains which variables or objects are contained in the class (data members) and which functions belong to the class (member functions). The class declaration does not include the actual code for the member functions.
- The *class definition* includes the code for all the member functions.

The class declaration has the following general format:

```
class classname
{
     declaration of data members
    declaration of member functions
};  // Don't forget the semicolon!
```

> **USING AND AVOIDING *STRUCT***
>
> Classes can also be declared as a structure using the keyword `struct`. This technique will be presented in Skill 20, but it is not encouraged—there is nothing to be gained.

The declaration of data members and member functions must be enclosed in braces, and there must be a semicolon following the closing brace! It is a common mistake to omit this semicolon. The compiler messages that result are very misleading and may not help you locate this error.

You need the following items in the class declaration presented in the general format above:

- The *classname* is any valid identifier of your choice. This will be the name you give to your new class of objects.
- The *declaration of data members* is just the declaration of all the variables or objects that will belong to each object of the class.
- The *declaration of member functions* lists what we call the *function prototypes* of the member functions. A prototype identifies a function, its result type, and the number and the types of arguments. The prototype is followed by a semicolon. You can mix declarations of data members and member functions in any order.

One Example of a *Clock* Class

As an example, let's examine one possible declaration of class `Clock`.

```
 class Clock //*************************  Clock
{
    long timestarted;
   public:
    float time();          // Returns the time elapsed
    void wait(float tsec); // Waits tsec seconds
    void reset();          // Resets
};
```

The first line presents the keyword `class` and the identifier (name) of the new class we are creating. Next, a set of braces (followed by a semicolon) encloses the declarations.

The `Clock` class has only one data member—the long integer `timestarted`. This is declared right after the opening braces, but could be declared anywhere inside the braces.

Then, the prototypes for the member functions appear: `time()`, `wait()`, and `reset()`. Notice that each prototype is essentially the same as the first line in the function itself. Each prototype contains the result type, the function identifier (name), and the types of arguments inside parentheses. You don't have to include an identifier for the arguments in the prototype (`tsec` could be omitted in the declaration for `wait()`).

The only thing unexplained in this declaration is the purpose of the word `public`? Hold on!

Class Definition

As we said before, the class definition contains the actual code for all the member functions in the class. This code is included in the program, essentially, in the same way the code is included for a nonmember function.

How do you specify that these functions are the member functions of your class? After all, C++ allows you to have a nonmember function `wait()` in addition to a member function `wait()` of class `Clock`.

Due to function overloading, several functions may have the same name. You must clearly state that the code you are providing refers to a member function of a given class.

You must attach each function you define to its class. This is done by preceding the function identifier by a qualification. The *qualification* is the class name followed by two colons, such as the following:

class name :: *function name* (*argument list*)

or, in our clock example:

```
Clock::wait()
```

Thus, the actual class definition for the class `Clock` could be as follows:

```
void Clock::reset()
{
    timestarted=timeGetTime();
}
float Clock::time()
{
   // Divide result by 1000 because we want
   //      result in seconds, not in 1/1000ths:
   return (timeGetTime()-timestarted)*.001;
}
void Clock::wait(float tsec)
{
   float willbe;
   if (tsec>60) tsec=60;     // Will not wait more than
                             //      1 minute!
   willbe=time()+tsec;
   while (time()<willbe);
}
```

You may try to type this class and use it. However, since there is already a class `Clock` in the software, the compiler will issue an error message. You can overcome this by using the name `clock` instead of `Clock`. You must also include the `mmsystem.h` header file.

Making Members Public, Private, or Protected

When you design a class, you may want the class members (either data or functions) to be either visible or not visible to other parts of the software. There are three possibilities.

Public

Public members can be accessed not only by your member functions, but also by any piece of program that can access the object. This is the case for the `reset()`, `time()`, and `wait()` member functions of class `Clock`. If an object of class `Clock` is declared, we want to be able to perform these functions with it.

Private

Private members can be accessed only by the member functions of the same class. It is not possible for other pieces of program to access a private member of an object. *Private* is the default. If nothing is declared about a member, it is assumed to be private. The `timestarted` variable is private—only the member functions of class `Clock` (`reset()`, `wait()`, `time()`) can use this variable.

Protected

Protected members can be accessed by member functions, but they can also be accessed by member functions of derived classes. We will see how to use these later.

The `public`, `private`, and `protected` keywords are enforced until another one of these keywords occurs in the listing. In other words, when a class declaration is started, it is assumed to be private. All members are assumed to be private until either `public` or `protected` is found. From then on, this new option is valid until another option is set. In the example of the `Clock` class, all the member functions are public, because the first one is preceded by `public`, and neither `private` nor `protected` can be found in the declaration.

ENCAPSULATION

The ability to isolate information from unauthorized interference called *encapsulation* in object-oriented programming. By making sure that data can only be manipulated by member functions that are associated with it, we can build more reliable and more maintainable software.

The software is more reliable because you can be assured that data is manipulated only as expected. It is more maintainable because, when modifications are needed, they are usually isolated to the object itself and are not scattered throughout the code.

Building Constructors

How can we be sure the clock will always be reset? The `Clock` class is still missing something. As you know, once a `Clock` is declared, it is automatically reset. This means that when a new `Clock` is created, the value of `timestarted` is set to the current time in the computer clock.

It must be possible for us to specify a special function that is automatically invoked every time an object is created. C++ allows us to do this by means of *constructor* functions.

The constructor function is very similar to other member functions, except for the following:

- Its name is the same as the class name. For example, the constructor for class `Clock` is `:Clock()`.
- It is automatically invoked when a new object is created. For example, when you have `Clock mine, yours;`, the constructor function `Clock::Clock` will be invoked twice—once to create `mine` and another time to create `yours`.
- It cannot be explicitly invoked in the program. For example, you cannot say `mine.Clock();`. The constructor is supposed to be invoked only once when an object is created.
- It does not have a return type (not even `void`). It is possible to have more than one constructor if the argument lists are different. For example, class `Box` has one constructor without arguments and one with an argument: `box x, y("Y:");` invokes the constructor without arguments for object `x` and invokes the constructor with one argument for object `y`. In this case, the box for `y` will automatically have a label that reads `Y:`.

Constructors must be public. They must be visible from any piece of program that can declare an object of that class, otherwise construction of the object will be impossible.

Initializing Objects with Constructors

Constructors are extremely useful for initializing objects. They provide us with an opportunity to prepare each object before it is put into use. Most classes of objects that you have used so far have a constructor. The clocks are reset, and the athletes are positioned one after the other. Other screen objects are assigned default coordinates and other attributes.

The complete declaration of class `Clock` is as follows:

```
class Clock //**************************  Clock
{
   protected:
    long timestarted;
   public:
    Clock();
    float time();          // Returns the time elapsed
    void wait(float tsec); // Waits tsec seconds
    void watch(float set); // Waits until time=set
    void reset();          // Resets
};
```

and the class definition should include the code for the constructor:

```
Clock::Clock()
{
   timestarted=timeGetTime();
}
```

Using Objects

When you build a class, you do not build one object of that class. You give an explanation of how all the objects of that class behave. Even if you are interested in using only one clock, you have to submit an explanation of the behavior of all possible clocks that you could use. Once the class `Clock` is built, you can use `Clock` objects in your program.

To use an object, all you have to do is to declare it. This is done in the same way you have been declaring athletes and clocks. The declaration consists of the class name, followed by spaces

and the identifiers (names) of the objects that you want to use. If you want to use more than one object, their identifiers should be separated by commas:

class name, object identifier;

or, in our examples:

```
Clock timer,mywatch;
athlete Sal;
```

The object declaration tells the compiler that you will be using in your program one (or more) object(s) of that particular class, and that you will be referring to that object by the identifier you provided. The compiler then sets apart some space in the computer memory, and makes sure that when the program execution reaches that stage, a call to the appropriate constructor is made to initialize the object.

Notice how similar the declaration for variables is:

type variable identifier;

For example:

```
float  x,y;
```

This reinforces the fact that there is a very strong resemblance between variables and objects, and also that there is an equivalent resemblance between types and classes. In this Skill, you will learn why we say that variables are just a limited form of objects, and for this reason, why we have referred to objects many times when we were referring to either objects or variables.

Accessing Object Members

Public members (functions and data) may be accessed by other parts of your program. To access a member, it is imperative that the object name precedes the member name as a qualifier. This is the way we have been accessing member functions of class `athlete`, in which the athlete's name is followed by a dot, and then the member function name. For example:

```
sal.ready();
```

invokes the member function `ready()`, which belongs to the object `sal`.

You can do the same thing with data members. For example, `Clock` objects have the data member `timestarted`, which can be accessed by preceding the data member name with the object name:

```
bigben.timestarted=mine.timestarted;
```

However, since `timestarted` is not public, it cannot be accessed by other parts of the program. It can only be accessed by member functions of class `Clock`.

Accessing Members in Member Functions

When you code member functions, the compiler understands that you are explaining how to manipulate each object of the class you are implementing. Since member functions can be invoked only for a particular object of that class, the compiler automatically understands that any reference, either to a member function or to a data member, is supposed to be tied to that particular object. This is why it is not necessary to qualify data members or member functions in member functions.

Creating a New Class of Objects

We are now ready to experiment with a new class—one that we will build ourselves. We will create a class of `wagon` objects. Wagons can be drawn on the screen and can move horizontally. Our wagon is built with a rectangle and two circles. Figure 16.3 shows our concept of a wagon.

The wagon class will have a constructor and a move() member function. The move() member function should make the wagon move on the screen. Obviously, we need a couple of variables to keep track of the coordinates on the screen, since the wagon will be moving.

One possible declaration for class wagon is as follows:

Fig 16.3 a wagon

```
class wagon
{
   Circle frontwheel,rearwheel;
   Square body;
   float x,y; // Coordinates
  public:
   wagon();
   void move();
};            // Never forget the semicolon!
```

There are three objects that function as data members: frontwheel and rearwheel (which are circles), and body (which is a square). There are also two floating point variables, x and y. There was no specification of whether these members are public. Therefore, they are assumed to be private. This is convenient, because only the move() member function and the constructor can access them.

Building a Wagon with Constructors

As we already know, the constructor will be invoked for each object created. The constructor assigns convenient sizes, colors, and locations for the objects that compose the wagon, and it sets initial coordinate values for *x* and *y*.

```
const int red=1,green=2;
wagon::wagon()
{
   x=60;                          // Initial coordinates
   y=200;
   frontwheel.resize(20);         // Assign size to wheels
   rearwheel.resize(20);
   body.resize(20,80);            // Make the body a rectangle
   frontwheel.color( red);        // Assign colors
   rearwheel.color(red);
   body.color(green);
   body.place(x,y);               // Place all parts
   rearwheel.place(x-25,y+10);
   frontwheel.place(x+25,y+10);
   body.show();                   // Show the wagon
   rearwheel.show();
   frontwheel.show();
}
```

```
void wagon::move()
{
   body.erase();                    // Erase from previous location
   rearwheel.erase();
   frontwheel.erase();
   x++;                             // Increment x
   body.place(x,y);                 // Place parts in new location
   rearwheel.place(x-25,y+10);
   frontwheel.place(x+25,y+10);
   body.show();                     // Show
   rearwheel.show();
   frontwheel.show();
}
```

The class above works, and you may try it as is. You may be wondering whether it is a good idea to define functions to `erase()`, `place()`, and `show()`. Yes, it is a good idea—but there is a better one!

Building a Wagon with the *Stage* Class

If we use one object of class `Stage`, we can insert the body and the wheels in the `Stage` object—we will have a much simpler program. This is shown in the program `c6train.cpp`:

```
#include "franca.h"
// Implementation of class wagon              c6train.cpp
// Part 6
// Version 2.0—uses a Stage
//
int const red=1,green=2;
class wagon
{
   Stage railroad;               // Stage included
   Circle frontwheel,rearwheel; // Declare wheels and body
   Square body;
   float x,y;                    // Coordinates
  public:
   wagon();
   void move();
};                               // Never forget the semicolon!

wagon::wagon()
{
   x=60; // Initial coordinates
   y=200;
   railroad.place(x,y);
   frontwheel.resize(20);        // Size the objects
   rearwheel.resize(20);
   body.resize(20,80);
   frontwheel.color( red);       // Color the objects
   rearwheel.color(red);
   body.color(green);
   body.place(x,y);              // Place the objects
   rearwheel.place(x-25,y+10);
   frontwheel.place(x+25,y+10);
   railroad.insert(rearwheel);  // Insert the objects in Stage
   railroad.insert(frontwheel);
   railroad.insert(body);
}
```

```
void wagon::move()
{
   railroad.erase();            // Erase previous
   x++;                         // Move forward
   railroad.place(x,y);
   railroad.show();             // Show current position
}
```

This new version alters the class declaration to include only an object of class `Stage`. The constructor did not change much, except that, instead of showing the body and wheels, it simply inserts them in the `Stage` object.

Making the *move()* Function Easier to Use

The `move()` function was simplified, because all the operations can be done with the `Stage` object, instead of each being repeated.

You can try our new class with the program below.

```
void mainprog()
{
   int i;
   wagon front;              // Declare front wagon
                             // Wagon constructor is invoked
   Clock station;            // And a clock
   for ( i=1;i<=80; i++)     // Loop
   {
     front.move();           // Move the wagon
     station.wait(.03);
   }
   wagon caboose;            // Declare one more wagon
                             // Wagon constructor is invoked
                             // Caboose is constructed only
                             //      after the previous loop
   for ( i=1;i<=200;i++)     // Loop
   {
     front.move();           // Move both wagons
     caboose.move();
     station.wait(.06);
   }
}
```

Accessing Class Members

Since the beginning of this book, we have been accessing member functions of the class `athlete`. To access a member function, qualify the function name with the object name followed by a dot. For example:

```
Sal.ready();
```

invokes the `ready()` member function of the object `Sal`, which belongs to class `athlete`. It is also possible to access data members in the same way. The data member must be preceded by the object name and a dot. For example, the program `c6train.cpp`, using the `wagon` class, can access the data members of this class. For example:

```
front.x=0.;                    // Sets x coordinate to zero
                               //      for first wagon
front.rearwheel.resize(15);    // Changes size of rearwheel
caboose.y=10;                  // Sets y coordinate to 10
                               //      for second wagon
```

However, if you try to do this, the compiler will issue an error message saying that these data members are not accessible. Why? Well, all the data members in class `wagon` are assumed to be private. If you insert the keyword `public` (followed by a colon) before the data members are declared, you may try again and see the results.

In this example, a wagon is always created at the coordinates (60,200). It may be a good idea either to allow the wagon to be placed somewhere else or to allow it to be created at a specified coordinate. Let's study some alternatives to this problem:

- Make the wagon's coordinates publicly accessible.
- Use arguments in the constructor.
- Have a member function change the coordinates.

Making the *x* and *y* Coordinates Public

If the coordinates are made public, you will be able to modify them after the object is created. By changing the coordinates, the wagon will be shown at a different location next time it is drawn. This seems like a pretty cool idea, doesn't it? Well, you will see that it is not!

This alternative is easy to implement—just include the following statement:

```
public:
```

before the declaration of `x` and `y` in the `wagon` class declaration. Then, you can change the wagon's position in the program by directly modifying the values of the coordinates of each wagon. For example:

```
front.x= 120;
front.y= 100;
caboose.x= 60;
caboose.y= 100;
```

will work if you change each coordinate before the wagon is moved. For example, we could have the following:

```
front.x=180;
front.move();
```

However, if you modify the coordinates after you have started moving the wagon (it is already drawn on the screen), you may have strange results. For example:

```
front.move();
front.x=200;
front.move();
```

Think about what could happen in this case—then go ahead and change the program to see the result. Do you like the result? Do you know why it happened?

Worse yet, these coordinates could be set to wrong values somewhere in the program:

```
caboose.x=z1;
caboose.y=z2;
```

You may ask, What would be wrong with this? Nothing, really—except that `z1` and `z2` could hold results of some strange computations, whose values could be –432 and 221,345, for example. How could you tell?

In any event, the result is undesirable. In the first case, the `move` member function erases the wagon from its current location, moves it one step along the horizontal, and then redraws it. If you change the coordinates, the old drawing of the wagon will not be erased properly. The `erase` function will erase the object using its current coordinate!

This is the reason for using `private` or `protected` in your class. You can make sure that only the member functions use these data members, and, since you are coding and testing them, you can guarantee they will work.

You could still argue that you are also coding the program, and, therefore, you know what you are doing to these data members. Don't do this. Never underestimate your ability to mess up software!

A bit of prevention is worth a megabyte of cure…

Finally, the most important argument in favor of keeping most data members nonpublic is that you should never build your software thinking you will be the only one to use it. Worse than this, never assume that you will make a program, use it, and then throw it away. The current trend is to reuse software. If you become a professional programmer, you will be building software for others to use. Software may be built in teams, and what you build will be used by somebody else to build something bigger. Every opportunity to avoid errors should be taken.

Using Arguments in the Constructor

The next alternative is to allow the constructor function to receive the coordinates as arguments. For example:

```
wagon::wagon ( int x0, int y0)
{
       x=x0;
       y=y0:
       ....
```

If the constructor is modified in this way (oops—don't forget to modify the constructor declaration in the class declaration, too), wagons can be placed anywhere when they are declared:

```
wagon  front (120,220);
...
wagon caboose(60,220);
```

The constructor without parameters is also known as the *default constructor*. It is always a good idea to include the default constructor in your class, even when you also provide a constructor with parameters.

Using a Member Function

You could consider yet another alternative—including a member function to place the wagon at a given location. This is reasonable and is much better then making the coordinates public. This member function can check the given coordinates to make sure they are valid.

This third alternative for implementing the class `wagon` has some advantages over the second one. However, I am still not happy with it! After studying inheritance in Skill 17, try to solve the same problem by deriving `wagon` from the class `Stage`, and see how much better you can do.

Try These for Fun…

- Modify the class `wagon` to show a picture resembling a locomotive. Include a couple of extra rectangles to represent the cabin and the chimney.
- Modify the class `Clock` into a new class (for example, `Clock1`), so that the member function `mark()` automatically resets the clock. In other words, if you want to wait for periods of 5 seconds, you could do the following:

```
Clock1 mytimer;
for (int i=1;i<=20; i++)
{
   mytimer.mark(5.);
}
```

- Create a class `pair` to consist of two athletes. `pair` should have the member functions `ready`, `up`, `left`, and `right`. Use this class to write a program that makes the pair perform a jumping jack.
- In implementing the `Clock::wait(float)` member function, I used a variable `willbe` to hold the current time. Try using `Clock::time()` to get the time.
- Remove the semicolon after the closing brace in the `wagon` class declaration. Observe the error messages that result.

Are You Experienced?

Now you can…

Make a new class by declaring it and defining it

Choose which members are public, private, or protected

Build constructor functions to initialize your objects

Create a new class of objects

Access data members and member functions in your objects

SKILL

SEVENTEEN

DERIVING CLASSES FROM EXISTING CLASSES

- Declaring and defining derived classes
- Inheriting members
- Understanding polymorphism

Making your own class of objects is not the most interesting application of classes. When it is possible, deriving a new class from an existing one is the real deal—you don't have to redo what you inherit, and you can modify whatever does not suit your taste.

We say that objects are smart because objects know how to do things that you tell them. For example, objects of class `athlete` know how to assume the positions `ready`, `up`, `left`, and `right`. They also know how to "say" something. In this Skill, you are going to learn how to create objects that are smarter than the ones you had before.

In the previous Skill, you were asked to modify the `wagon` class in a few exercises in the "Try These for Fun…" section. To work through these exercises successfully, you have to do one of the following:

- Take the old code, remove parts that are no longer needed, and insert new parts that are needed. In this case, you have only one piece of code implementing the new class. There is no way to use the old class.
- Copy the old code, and then make alterations as needed. In this case, you have two versions of the same class. You may even consider using a different name for the new class. You are able to use objects of both classes in the same program.

The second alternative is slightly more convenient, because the old class remains usable and unaltered. If it remains unaltered, you are sure that no new errors can be inserted in the code. But now you have two pieces of code to fix when you have a problem! Another concern is that if you use the two classes in the same program, the common code (the unaltered part) will be duplicated.

To have available two (or more) pieces of code that do essentially the same thing is a real pain in the neck for software management. It may not seem so to you at this time, but since software has to be updated—not only to be corrected, but in response to external factors—it will be necessary to perform the same maintenance in all pieces of software. It is definitely preferable to use, or better yet, to reuse a unique piece of code.

Deriving Classes

Object-oriented programming has a special feature to help you reuse software. When you want to use a class of objects that is very similar to an existing one, yet you want it to operate slightly differently, you can *derive* a new class from the existing one. In C++, the original class is called the *base class;* the one you derive from it is called the *derived class*.

A special kind of `wagon` that looks like a locomotive and a special kind of `Clock` that resets itself after each waiting period are good possibilities for using derived classes. This would lead us to new classes: `locomotive`, which is a special kind of `wagon`, and `timer`, which is a special kind of `Clock`.

Each object of the derived class can still be regarded as an object of the base class from which it was derived. A locomotive can still be regarded as a wagon, and a timer can still be regarded as a clock. In fact, a locomotive is just another wagon composing the whole train.

The good thing about deriving classes is that anything that already works in the base class will be automatically *inherited* by the derived class. You never need to duplicate anything that previously existed! You only have to worry about the special features you want implemented in your new, derived class.

For example, to implement a `locomotive` class, you don't have to specify how to draw the wheels or the wagon itself. There is also no need to explain how to move it. All you have to specify is how to draw the cabin and the chimney, which were not present in the base `wagon` class.

Similarly, for the `timer` class, all you have to do is to write the `mark` member function, so that it will reset the clock every time.

Creating Derived Classes

To derive a class from another, the steps are similar to the ones you took to create a new class:

1. Declare the class—declare all the members that do not exist in the base class (member functions that will replace existing member functions in the base class must also be declared). When you declare a derived class, explicitly mention that this class is derived from a given base class.
2. Define the class—write the code for the new member functions.

Declaring the Derived Class

When you declare a derived class, you must state that it is a derived class, and you must identify the base class from which it is derived. This is done right after we give the name of this new, derived class. The syntax is as follows:

class *derived class name* : public *base class name*

{

member declaration

}; // Never forget the semicolon!

Notice the addition of the colon, the keyword `public`, and the base class name that identifies the base class from which we are deriving.

The following could be the declaration for the `timer` class:

```
class timer: public Clock
{
   void mark (float time);
};
```

Defining the Derived Class

The definition of the new member function could be as follows:

```
void timer::mark(float time)
{
   watch(time);
   reset();
}
```

The new class `timer` will still be able to perform all the member functions that already existed in the base class. The functions `reset()`, `time()`, `watch()`, and `wait()` are inherited from the base class. You don't have to specify anything more about these functions.

Actually, you may not even know how they work! This is the wonder of inheritance—you get something without having to work for it.

Classes and Objects in the Real World

The classes and objects that are used in object-oriented programming are concepts borrowed from the real world. Objects are the things we see around us. What things? Well, anything that exists—any of your colleagues, a piece of furniture, a car driving down the street, the stoplight at the intersection, even the birds flying overhead! These are all examples of objects (and don't forget the objects you see on your computer screen).

Of course, you may have realized that some of these objects are similar. They may look alike, or they may have similar characteristics. Take your colleagues, for example. Although each colleague is an individual, each person shares common characteristics with every other person that distinguish them from stoplights, furniture, or birds.

We may say that objects that have the same characteristics belong to the same class. We can readily identify some classes:

- The class of colleague objects
- The class of stoplight objects
- The class of furniture objects
- The class of bird objects
- The class of Screen Objects

The class is the set of all objects, not one object in particular. So, there is a class of stoplight objects, but the stoplight down the street is one object. Each colleague is one object, as is the desk you use.

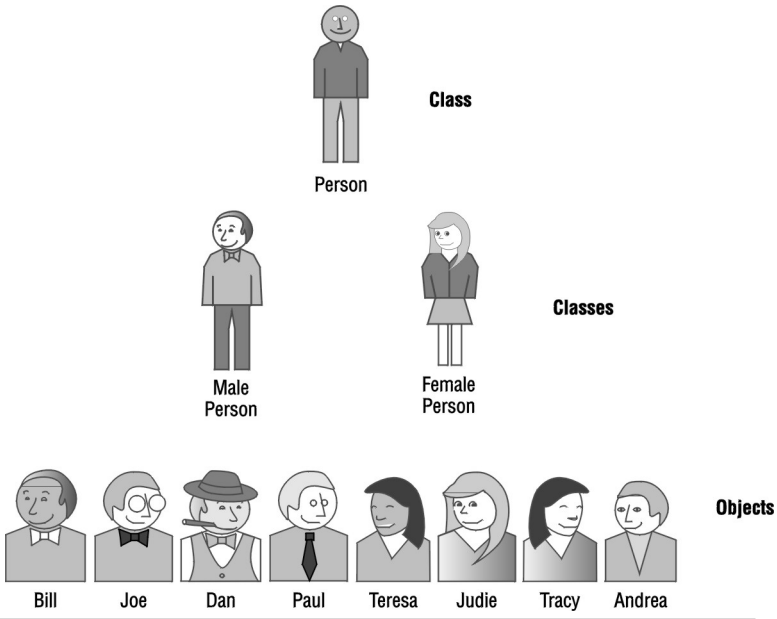

Fig 17.1 Classes and Objects

Furthermore, classes can be subdivided to help you understand the world better. For example:

- You can split the class *furniture* into desks, chairs, lamps, beds, etc.
- You can classify birds into many species.
- You can classify your colleagues.

Parent Classes and Child Classes

Figure 17.1 shows the class *person*, which has the descendents *male person* and *female person*. It then shows a few objects of these classes in the bottom row.

There is more to this simple picture than what meets the eye at first. If you look carefully, you will notice that all the pictures "descend" from the one at the top. The picture illustrating the class *person* defines the basic body used in the next classes.

From the class *person* we derived two other classes, *male person* and *female person*. Keep in mind that both *male* and *female* are also *person*. Both classes have all the basic attributes of a

person, except that when you designate a person as *male* or *female*, you give them additional attributes that are undetermined in the class *person*. In our overly simplified world, the *male* has pants, and the *female* has skirts and longer hair.

Remember that a derived (or descendent) class has all the attributes of the base (or parent) class. Therefore, a *male person* is also a *person*.

Notice also that a derived class designates a *special kind* of class within the existing base class. *Male person* is a special kind of *person*. *Female person* is another special kind of *person*. Similarly, a `Circle` is a special kind of `ScreenObj`.

Objects—Particular Instances of Classes

Person, *male person*, and *female person* are still vague. There are many people who can be denoted by those class names. If you want to refer to a specific individual, you have to point out one particular *person*—a particular *instance* of that class.

Again, our simplified example (shown in Figure 17.1) shows a group of individuals at the bottom. They are specific. We are now talking about Bill, Joe, Dan, and Paul, as well as about Teresa, Judie, Tracy, and Andrea. Bill is not a class! Bill is one specific guy who happens to be a *male person*. Bill is an object that belongs to the class *male person*. Joe, Dan, and Paul are also objects that belong to the class *male person*. Similarly, Teresa, Judie, Tracy, and Andrea are objects that belong to the class *female person*. All these objects also belong to the class *person*.

Notice that in our simplified world, everybody looks the same. All the pictures of guys were built using the basic *male person* picture—we just dressed them in different clothes. The same is true for the ladies.

In your programs, you can also use objects to represent the real world. You have used objects of class `athlete`, Sal and Sally. You have used an object of class `Robot`, Tracer. You have used circles, squares, boxes, and clocks.

Special Kinds of Screen Objects

The idea of deriving a class from another one was exploited extensively with Screen Objects. `ScreenObj` was a class that allowed you to place, resize, erase, show, and color objects on the screen. The class `Box` is derived from `ScreenObj`. In other words, `Box` is a special kind of `ScreenObj`. You can do anything with a `Box` that you can do with a `ScreenObj`. However, you can also label and you can "say" something with a `Box`. The idea is very simple: if you want to create a *special class* that is derived from an existing class, you have to do the following:

- Give a name to the new class, and indicate the existing class from which it is derived. For example, the new class could be `Box`, and we would indicate that it is derived from `ScreenObj`. The class could also be `athlete`, and we would indicate that it is derived from `Stage`.
- Declare any variables or objects that will exist in the new class that did not exist in the base class. Each object of the class `Box` may have data, such as the label and the message to be printed. These data did not exist in a plain `ScreenObj`. Each object of class `athlete` will have several objects: head, trunk, left arm, right arm, left leg, and right leg. These objects did not exist in a plain `Stage`.
- Declare and write the code for any functions that will exist in the new class that did not exist in the base class. Each object of the class `Box` must know how to `label` and `say`. Each object of the class `athlete` must know how to assume `ready`, `up`, `left`, `right`, etc.

Objects of the new, derived class possess all the data and functionality of the base class, as well as the new data and functions that were specifically added when the derived class was created. We only have to write programs for the functions that we are adding. We can completely reuse all the software that existed.

Figure 17.2 shows the relationship between a base class `ScreenObj` and the derived classes `Circle` and `Box`. An object of the class `ScreenObj` contains the data and functions listed in the inner, *ScreenObj* rectangle. An object of the class `Box` contains the data and functions listed in the outer, *Box* rectangle.

Notice that the *Box* rectangle contains the *ScreenObj* rectangle. Therefore, a `Box` contains its own data and function members, as well as those inherited from the `ScreenObj` class. Similarly, objects of class `Circle` contain all the members enclosed in the *Circle* rectangle, which, of course, also includes the *ScreenObj* rectangle.

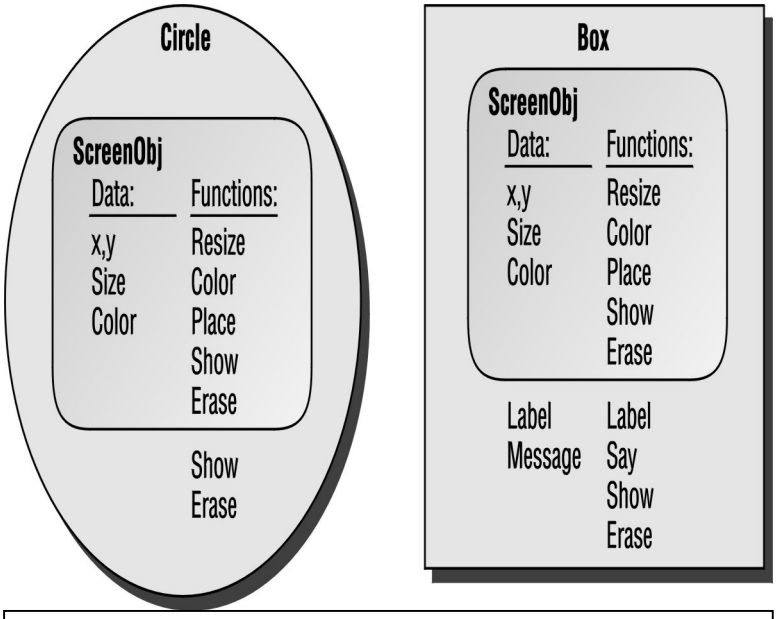

Fig 17.2 ScreenObj, Circle and Box

Inheriting Characteristics from Base Classes

Another way to represent the relationship between classes is to use a hierarchical diagram, as shown in Figure 17.3. In this diagram, each box represents a class of objects. The classes that are used as bases from which to derive new classes are shown above, and are linked to the derived classes below. For example, the class `ScreenObj` is a base for the classes `Circle`, `Square`, `Box`, and `Stage`. The class `Stage` is a base for the class `athlete`. Also, it is not shown, but the class `athlete` is a base for the class `Robot` (no wonder they look so much alike).

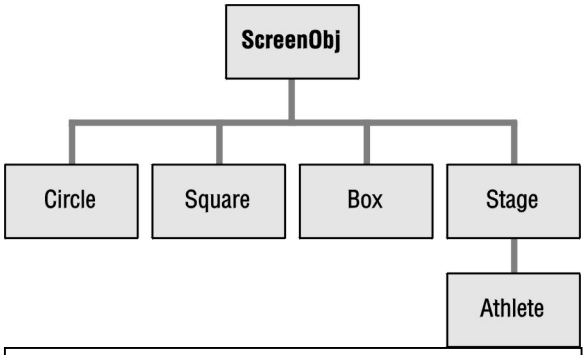

Fig 17.3 Base and derived classes

C++ nomenclature includes base and derived classes. It is also common to refer to parent and child classes, or to refer to ancestor and descendent classes (instead of base and derived classes).

A derived class maintains all the characteristics of the base class. This is called *inheritance*. The derived class has either additional data or additional functions (or both).

An object belonging to a derived class may still be considered as an object of the base class: a `Circle` is a `ScreenObj`, because you can use a `Circle` anywhere you can use a `ScreenObj`. The opposite is not true, however.

Although every `Circle` is a `ScreenObj`, a `ScreenObj` is not necessarily a `Circle`.

Replacing Inappropriate Inherited Functions

Compared to `ScreenObj`, `Circle` does not have any additional data members. However, you may wonder why we repeat the member functions `show` and `erase` in each class. After all, don't circles and boxes inherit these member functions from the base class `ScreenObj`?

They do, but isn't the procedure for drawing a circle different from that of drawing a box? What kind of shape will be drawn if we use the `show` function of a Screen Object? There may be cases in which the inherited function is not appropriate. You must supply another function to perform the equivalent action.

In this example, although a Screen Object has coordinates, size, and color, it does not have a defined shape. Therefore, the `show` function cannot show anything! Boxes, circles, and squares have a shape and can be drawn, but the procedure that draws a box is not the same as the one that draws the circle or the square. Each of these classes will have a different `show` function.

But what happens to the inherited `show` function? It *is* inherited, isn't it?

Yes, it is. However, it is automatically understood that if you use an object of class `Circle` and you issue `show`, the `show` that is used is the one defined for the class `Circle`. This is the essence of *polymorphism*, which we will discuss later.

Even though the `show` function for `ScreenObj` is fake, it must exist. If a function prototype is not included in the class declaration, the compiler does not allow any objects of that class to use that member function.

The *athlete* Class

The class `athlete` descends from `Stage`, which, in turn, descends from `ScreenObj`. All objects of class `ScreenObj` have a set of variables *x*, *y*, and *z* that represents the screen coordinates of the center of the object. Although there are only two dimensions to the screen, the *z* coordinate may be used to place an object behind another. The following is the declaration of data members for class `ScreenObj`.

```
protected:
  float x,y,z;              // Screen coordinates of object
  int colorbrush,colorpen; // Inside color and contour
  float size,length;
```

Since all these data members are *protected*, they are not only inherited, but also accessible from the descendent classes, such as `Stage` and `athlete`.

Deriving More Classes

In our next example, we will derive two new classes from `athlete`:

- `runner` will contain objects similar to athletes, but it will have the ability to "run." Runners will move their legs and arms while stepping to a neighboring location on the screen.
- `skater` will also have the ability to "run." However, since it will run while wearing skates, the skater will slide over a few neighboring locations before switching from one leg to the other.

The most tedious task in this implementation is the simulation of the arm and leg movements. You could do this by studying how the athlete is built, but I will save you some time and give you a couple of functions, `stepleft` and `stepright` that will do the trick.

Figure 17.4 shows athletes stepping right and stepping left. The animated sequence—ready, stepright, ready, stepleft—as shown in Figure 17.5, will create the illusion that the athlete is actually moving their arms and legs.

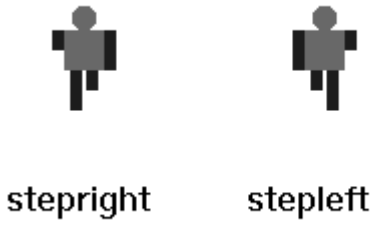

Fig 17.4 Stepping right and left

Fig 17.5 Sequence for running

The function `stepright` draws the athlete with the right leg and the left arm raised (from the viewer's vantage point). The function `stepleft` does the same with the left leg and the right arm. Both functions can receive a floating point argument that expresses the time, in seconds, that we want to hold the picture in the given position. Here are the functions that can be found in `c6run.h` (the values of `armsize` and `armwidth` are already defined in franca.h. They are included in the initial comment so you remember what they are):

```
// const int armsize=20,armwidth=6;
void runner::stepright(float time)      // c6run.h
{
   Clock any;
   erase();
   leftarm.resize(armsize/2,armwidth);
   leftarm.place((x-(armsize+armwidth)/2.),y-armsize/4);
   rightleg.resize(armsize/2,armwidth);
   rightleg.place(x+(armsize/2.-armwidth),y+3*armsize/4);
   show();
   any.wait(time);
}
void runner::stepleft(float time)
{
   Clock any;
   erase();
   rightarm.resize(armsize/2,armwidth);
   rightarm.place(x+(armsize+armwidth)/2,y-armsize/4);
   leftleg.resize(armsize/2,armwidth);
   leftleg.place(x-(armsize/2-armwidth),y+3*armsize/4);
   show();
   any.wait(time);
}
```

Athletes That Run and Skate

Since we know how to draw the athlete in these new positions, the `run()` function essentially consists of the following steps:

- Step left
- Ready
- Move one position
- Step right
- Ready
- Move one position

Remember to erase the pictures immediately before you move them, because the `erase` function simply redraws the picture in white at the current location. If you move the picture and then erase it, a piece of the picture that was drawn in the old location will not be erased.

It is very simple to move the runner. Since the runner is an athlete (and, consequently, a `Stage` object), we can use the `place()` member function to place the runner anywhere we want. Moreover, since we already know the *x* and *y* coordinates, we can easily figure out where the next location will be.

In this implementation, we are making the runner go only from the left to the right of the screen, which means that the *x* coordinate will increase with each movement.

Here is one implementation of the `run()` member function:

```
void runner::run()
{
   stepleft(.1);
   erase();
   ready(.1);
   stepright(.1);
   erase();
   ready(.1);
   erase();
   place(x+1,y);
   ready(0.);
}
```

Make sure you do not update the value of the screen coordinates yourself. The `place()` member function will update the coordinates of all the objects contained in that `Stage` object by computing the difference between the new coordinates (given as arguments to `place()`) and the previous coordinates (the values of *x* and *y*).

If you update the value of *x,* or use `x++` as the parameter, the new and old coordinates will be the same, and no movement will take place.

The declaration of the class `runner` is as follows:

```
class runner: public athlete
{
  protected:
   void stepleft(float);
   void stepright(float);
  public:
   void run();
}; // Never forget the semicolon!
```

It is useful to keep the `stepleft` and `stepright` member functions `protected`, because it will make it impossible for a common piece of program to use these functions, but it will allow member functions of descendent classes (such as `skater`) to use them.

The declaration of the class `skater` can be as follows:

```
class skater:public runner
{
 public:
  void run();
};
```

Overriding an Inherited Function: Polymorphism

Notice that `stepleft` and `stepright` are not declared again in the example above. They are inherited from the base class `runner`. You may think it is odd that we are declaring a member function with the same name as one that is already inherited! What will happen in this case? When you tell a skater to run, the object will use the `run` member function that was specifically declared for the class `skater`, and will not use the inherited one. Essentially, this is what we call *polymorphism* in object-oriented programming. You can specify a different way of handling a given request (`run()`), according to the object class that you are using.

The `run()` function for the skater simulates the skater sliding for a distance while alternately standing on each foot. Here is one implementation:

```
void skater::run()
{
  for(int i=1;i<=5;i++)// Move 5 steps
  { // Standing on left foot:
    stepleft(.1);
    erase();
    place(x+1,y);
    ready(0.);
  }

  for(i=1;i<=5;i++)    // Move 5 steps
  { // Standing on right foot:
    stepright(.1);
    erase();
    place(x+1,y);
  }
  ready(0.);
}
```

You can make both athletes run by using a program such as the following program:

```
void mainprog()
{
  runner jane;
  jane.place(20,80);
  skater julia;
  int k;
  for (k=1;k<=10;k++)
  julia.run();
  for (k=1;k<=10;k++)
  jane.run();
}
```

What if you want to make the skater run like a runner? She should be able to do this, since the skater is still a runner, after all. All you have to do is to use the `run()` member function, which the skater is entitled to as a member of the class `runner`, instead of using the standard member function belonging to the class `skater`. To do this, qualify the `run()` function with the class name `runner`. In this example, it would suffice to include the following statements:

```
  for (k=1;k<=10;k++)
  julia.runner::run();
```

to have Julia run as a runner.

Polymorphism Defined

Polymorphism is a term originating from Greek. It means "many shapes." Objects that share common ancestry can assume many shapes, and still be treated the same. Notice that both the runner and the skater can `run`. Yet, the way a skater runs is not the same as the way a runner does. The nice thing is that you don't have to constantly worry about this. You can just tell either one of them to run, and they will rush off in whichever way they know.

This concept was already explored with the Screen Objects. All of them could be shown, erased, and placed anywhere. Nevertheless, the way we show a square is not the same way we show a circle, an athlete, a robot, a stage, etc. Objects of the class `ScreenObj` have many shapes indeed!

The *Stage* Class and Polymorphism

`Stage` objects are a particularly interesting case, since a `Stage` object may contain several other Screen Objects, including other `Stage` objects. For example, you can insert an athlete (whose attributes are inherited from the `Stage` class) and a runner (whose attributes are also inherited from the `Stage` class). When the `Stage` object is told to show itself, it will know precisely how to show each part of itself. However, we have not yet explored fully the polymorphism capability.

For example, suppose that you have a function that takes a runner as an argument.

```
void march(runner volunteer)
{
   volunteer.run();
}
```

If you have the following program:

```
runner smith;
skater yamaguchi;
march (smith);
march (yamaguchi);
```

you might expect that it will cause `smith` to run like a runner and `yamaguchi` to run like a skater. This will not happen, though. The function `march()` knows only that a `runner` will be used. When this function is compiled (translated into machine language), there is no information specifying which runner will be used.

It is only when the program is executed that the function will receive either `smith` or `yamaguchi`. However, at that point it is a bit too late! The function is already translated into machine language, and it thought that using the `run()` function that was defined for a `runner` was the appropriate thing to do.

To postpone the decision to choose the kind of `run()` function to use, we can specify that the runner's `run()` function is a virtual function. This is done by simply including, in the declaration of the `runner` class, the keyword `virtual` in the `run()` function prototype. For example:

```
virtual void run();
```

instructs the compiler to avoid sticking to the `runner` class's `run()` function, allowing it to be substituted later by any other version. Of course, this makes your program a little larger and slower.

Try These for Fun…

- Modify the `skater::run()` member function so that the skater has to take some steps (running as a runner) before being able to actually skate.

- Since both skaters and runners are athletes, they should be able to behave like athletes, as well. Write a program that declares a `runner` and a `skater`, and have them perform a jumping jack, then run, and then do another jumping jack.
- Although a runner is an athlete, an athlete is not a runner. If you declare an `athlete` and order the athlete to `run()`, what will happen?
- The function `jumpjack` you developed that takes an argument of class `athlete` can also take `runner` and `skater` as arguments. Try this and see how it works. C++ is severe about making a correspondence of type between the parameters defined in the function and the actual arguments used in the function call. Why is it that you can use the `jumpjack` function with a runner, while it was designed to be used with an athlete?
- Write a function `goaway(runner someone)`. This function should make the `runner` that was passed as an argument run 10 steps. Use this function with an object of class `runner`, and with an object of class `skater`. Notice the results.
- Include a constructor function for the class `skater` that includes a skateboard at the skater's feet. The skateboard can be easily constructed with a rectangle and two circles.

Since the skater descends from `Stage`, you can simply `insert()` the rectangle and the two wheels.

- The `wagon` class as implemented does not allow the user to place, erase, and show a wagon. This is because `wagon` is not a descendent of `ScreenObj`. Implement another version of the `wagon` class by deriving it from `Stage`. In this case, you don't have to declare a `Stage` object—the class is already a `Stage` object.

Are You Experienced?

Now you can…

Derive a new class from an existing class

Inherit data and functions from a base class

Manipulate objects in the same class hierarchy as if they were alike

SKILL

EIGHTEEN

DEVELOPING MORE ADVANCED APPLICATIONS

- Completing a short project—an improved point-of-sale terminal
- Choosing a programmer-user interface
- Choosing an end-user interface
- Choosing and implementing classes
- Completing another short project—a satellite simulation

The ability to develop applications is the essence of the programmer's profession. This ability has been continuously developed in most Skills in this book. In Skill 18, you will continue to improve it by completing two short projects:

- Developing an improved point-of-sale terminal using classes
- Developing a satellite simulation using classes

Short Project 1—an Improved Point-of-Sale Terminal

You will now learn how to transform the point-of-sale terminal you developed in Skill 9 into an object. The goal is to use as many of the features as possible that were discussed in Skill 9 (using the `c3sale2.cpp` program).

The functionality of the terminal will be the same as the previous terminal's functionality. All I want to do now is to show you how to transform the terminal into an object. This will allow you to work on interesting expansions in the next Skills. You may benefit from looking back to Skill 9, and from running the `c3sale2.cpp` program once more.

Choosing the Programmer-User Interface

As you progress in your programming career, more people will use the programs that you develop. Some people may use your software as end users. For example, a clerk at a store using your terminal program is an end user. This person does not know how your program was built. All he or she cares to know is how to hit the right keys to do his or her job. However, there is another kind of user of whom you must be aware.

The Programmer User

In a world in which software is extremely complex, programmers use programs that were developed by other programmers to produce better software. You have been using the programs in `franca.h`, which I developed. You are a programmer user of my software. However, you use this software to help you produce your own software, which may have an end user who knows nothing about programming.

When you design software for programmer users, you must also exercise care to make it simple to use.

The current project consists of creating a class of point-of-sale terminals, so an object of this class can be used in a program to generate a final product—the terminal.

In this case, you will implement the class, and you will be the one to use it. You will be your own programmer user. However, this may not always be the case!

Interfacing

There may be many ways to implement the terminal as a class. One possibility is to have a terminal that is only created and told to operate. This means that the main program, besides creating the terminal, only has to issue a call to one function to put it to work.

The main function could look as follows:

```
void mainprog()
{
  terminal cashregister;
  cashregister.operate();
}
```

This will indeed be a very simple interface for the programmer who uses your class.

A class that does everything by itself, like this one, may not be the best idea. You may consider including some operations that allow the programmer to manipulate the object with greater flexibility. For example, you could provide member functions that allow the programmer to write to the boxes or to process one customer at a time.

Choosing the End-User Interface

The end-user interface will be the same one used in `c3sale2.cpp`. It is a good idea to keep the user interface as long as the customers are happy with it. Anytime you change the user interface—move a box from one place to another, ask a question in a different way, etc.—the user will spend some time to get used to the changes. Sometimes the user will not like the changes!

Declaring and Defining Classes

The `terminal` class should contain most of the variables and objects that were present in the previous program. This could lead to the following declaration:

```
class terminal
{
 protected:
  float tax,amount,price,saletotal,change;
  Box Price,Cur_Total;
  Box Saletotal,Amount,Change;
  void items(); // Process items in a sale
 public:
  terminal();
  void operate();
};
```

Notice that in the `c3sale2.cpp` program, you could initialize the boxes. In this case, you cannot initialize the boxes in the class declaration. This task has to be left to the constructor.

No initial values can be set in a class declaration.

The Constructor Code

The constructor code is relatively simple. The only reason it looks a little long is that the boxes have to be positioned and provided with a label.

```
terminal::terminal()
{
  tax=ask("Enter the sale tax %")/100.;
  Price.place(450,100);
  Price.label("Price:");
  Cur_Total.place(450,140);
  Cur_Total.label("Subtotal:");
  Saletotal.place(450,180);
  Saletotal.label("Total Sale:");
  Amount.place(450,220);
  Amount.label("Amount Tendered");
  Change.place(450,260);
  Change.label("Change:");
}
```

Figure 18.1 shows these boxes after processing a sale.

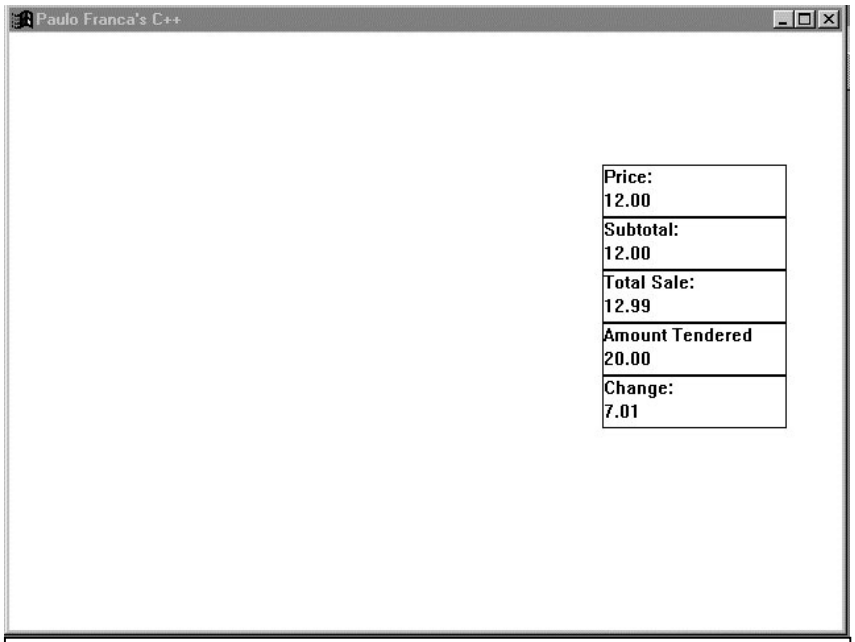

Fig 18.1 Boxes after sale processing

The *operate()* Member Function

The `operate()` member function is the function that actually makes the terminal work. You may notice that this function closely resembles what was done in the previous program. A loop handles the sale to each customer. After each sale, the total is displayed and the change is computed.

To simplify the code, the loop that handles each item in a sale is handled by another function `items()`. Still, this is just like the `c3sale2.cpp` program. However, notice that the programmer user is not supposed to invoke the `items()` member function. This function was designed to be used internally only. For this reason, this member function is not public.

Here is the code for the `operate()` function:

```
void terminal::operate()
{
   float tendered,change;
   for(;;)
   {
     saletotal=0.;
     Price.say(" ");
     Cur_Total.say(" ");
     Saletotal.say(" ");
     Amount.say(" ");
     Change.say(" ");
     items();
     Saletotal.say(saletotal);
     amount=ask("Enter amount tendered:");
     Amount.say(amount);
     change=amount-saletotal;
     Change.say(change);
     if(!yesno("Another customer?"))break;
   }
}
```

Notice that all the values and the boxes displaying them are cleared at the beginning of each sale. Then, the `items()` member function handles all the items in the sale. After doing this, all the boxes are updated with the pertinent information.

The program above uses two very similar names for different things: there is a box named `Change` (capitalized) and a variable named `change` (lowercased).

The *items()* Function

The `items()` function serves a purpose similar to that of the `getitems()` function used in `c3sale2.cpp`. It adds all the items purchased by the same customer.

The code for the `items()` function is as follows:

```
void terminal::items()
{
  for(;;)
  {
    price=ask("enter the price:");
    saletotal=saletotal+price;
    Price.say(price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
  saletotal=saletotal*(1+tax);
  Saletotal.say(saletotal);
}
```

IMPROVEMENTS

The most relevant improvement to this terminal would be to replace input from a price with input from a code. This is what most present-day point-of-sale terminals do. Currently, you don't have the means to make this improvement. Further versions of this project will be presented in upcoming Skills.

The Complete Program Listings

The complete code for this project can be found in `c6term.h` and `c6term.cpp`.

The Main Program *c6term.cpp*

Here is the code for `c6term.cpp`:

```
#include "franca.h"
#include "c6term.h"
void mainprog()
{
  terminal cashregister;
  cashregister.operate();
}
```

The Header File *c6term.h*

Here is the code for `c6term.h`:

```
#ifndef C6TERM_H
#define C6TERM_H
#include "franca.h"
class terminal
{
 protected:
  float tax,amount,price,saletotal,change;
  Box Price,Cur_Total;
  Box Saletotal,Amount,Change;
  void items(); // Process items in a sale
 public:
  terminal();
  void operate();
};
terminal::terminal()
{
  tax=ask("Enter the sale tax %")/100.;
  Price.place(450,100);
  Price.label("Price:");
  Cur_Total.place(450,140);
  Cur_Total.label("Subtotal:");
  Saletotal.place(450,180);
  Saletotal.label("Total Sale:");
  Amount.place(450,220);
  Amount.label("Amount Tendered");
  Change.place(450,260);
  Change.label("Change:");
}
```

```
void terminal::items()
{
  for(;;)
  {
    price=ask("enter the price:");
    saletotal=saletotal+price;
    Price.say(price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
  saletotal=saletotal*(1+tax);
  Saletotal.say(saletotal);
}

void terminal::operate()
{
   float tendered,change;
   for(;;)
   {
     saletotal=0.;
     Price.say(" ");
     Cur_Total.say(" ");
     Saletotal.say(" ");
     Amount.say(" ");
     Change.say(" ");
     items();
     Saletotal.say(saletotal);
     amount=ask("Enter amount tendered:");
     Amount.say(amount);
     change=amount-saletotal;
     Change.say(change);
     if(!yesno("Another customer?"))break;
   }
}
#endif
```

Short Project 2—Satellites

The program `c5stars.cpp`, developed as a project in Skill 15, will now be reviewed and improved with the use of objects. Indeed, the circles were used to represent a star (the Sun), a planet (Earth), and a satellite (the Moon).

Generally speaking, these objects orbit other objects at a particular distance and angular velocity. If you think abstractly instead of using the usual nomenclature that differentiates between stars, planets, and satellites, you may realize that all of them can be treated like satellites.

Indeed, Earth, as well as the other planets in our solar system, is a satellite orbiting the Sun. You can consider that the Sun itself is orbiting any distant body at speed 0, or that it is orbiting itself. This abstract thinking leads to the idea of creating a new class of objects—satellites.

Satellites can be used to simplify the `c5stars.cpp` program, and can also be used in other applications.

A satellite can be represented by a circle that moves in a circular pattern around a specific point or, to be more interesting, around another satellite (including itself). Besides all the usual features of a circle, satellites have the following features:

- A planet to orbit. This planet gives the coordinate of the point they are orbiting.
- A distance from that planet. This is the radius of the orbit.
- An angular velocity for the movement.

- A current angle in the orbit. This angle and the radius constitute the polar coordinates of the satellite, relative to the planet.

In addition, satellites should know how to move from one place to the next place as time passes. However, instead of using a time-dependent equation to compute the position, you can define a function `move` that will be invoked for each drawing frame (every 30th of a second). This function computes the coordinates of the satellite after each frame time elapses.

Designing the Satellite Class

Here is the declaration of the `satellite` class:

```
class satellite: public Circle
{
  protected:
   float xplanet,yplanet;
   float radius,omega,wspeed;
  public:
   satellite();
   move();
};
```

Data members include the planet's coordinates (notice that you don't need the planet—you only need to know where it is), the orbit's radius (`radius`), the current angle (`omega`), and the angular velocity per unit of time (`wspeed`).

There are a few design choices concerning the angular velocity. Ultimately, you need to know how many radians the satellite moves per time frame (a 30th of a second). However, you may choose to allow a user of your class (including yourself) to specify it as degrees per second, even though you convert this value and store it as radians per time frame.

Using a Constructor

A constructor function is always desirable, so we can color, size, place, and perform other initialization tasks with the object. The `move` function, discussed in an earlier Skill, should be called to move the satellite to a new location in the orbit after the time frame has elapsed.

It is clear that you could use a function such as `move(t)` to determine the current time and to compute the position at this time.

Unfortunately, the class declaration above is not appropriate, yet. How will we set values for the data members?

One alternative is to allow the constructor to take these values as parameters. This would work, but once a value is set, it cannot be changed unless one of the following items is true:

- The data member is public.
- There is a member function that allows you to change the values.

Even though you may think that some of the data members, such as the radius, the coordinates of the planet, or the angular velocity, will never change, you may be amazed by what you can accomplish with a little flexibility.

Member functions can be included to change the following items:

- The coordinates of the planet
- The radius
- The angular velocity

Specifying a Coordinate System for the Satellites

Finally, as you may have noticed in `c5stars.cpp`, it will come in very handy to convert from polar coordinates to cartesian coordinates. Include a `polarxy()` function in the class. Here is the declaration proposed for this design:

```
class satellite: public Circle
{
  protected:
   float xplanet,yplanet;
   float radius,omega,wspeed;
   void polarxy(float r,float theta,float x0,
                float y0,float &x,float &y);
  public:
   satellite();
   void center(float x,float y);
   void dist(float rad);
   void angle(float ang);
   void speed(float ws);
   void move();
   void center(satellite &another);
};
```

There are two `center()` functions that allow you to change the planet's coordinates. These functions allow you to specify that a planet revolves around a given point (*x*,*y*) or around another planet.

In this case, the `polarxy()` function can be modified so no arguments are needed. This may be a good idea, but it is left unmodified, so we reuse the same code.

Implementing the Class

Once the class declaration is ready, the code for the functions can be developed.

The Constructor Code

Here is the constructor code:

```
satellite::satellite()
{
   color(2);
   xplanet=yplanet=0;
   radius=0;
   omega=wspeed=0;
}
```

There is nothing special about this constructor. Notice that the satellite's color can be modified at any point, since the `color()` function is public for the circles.

Changing the Center

The functions to change the planet's coordinates are as follows:

```
void satellite::center(float xc,float yc)
{
   xplanet=xc;
   yplanet=yc;
}
```

```
void satellite::center(satellite &another)
{
   center(another.x,another.y);
}
```

Do you see anything interesting in this code? Why does the second function use the *x* and *y* coordinates of the other planet? Aren't these coordinates protected?

Indeed, they are! However, protected members can be used by member functions of the same or derived class. Since the second function is a member function for `satellite`, which is derived from `Circle`, it is possible to access the coordinates. Not only the coordinates of this satellite are accessible, but the coordinates of any other satellite that is used inside the member function are accessible.

It makes the use of satellites much more convenient to include the second version of the `center()` function. It is definitely easier to say

```
Moon.center(Earth);
```

than it is to determine Earth's coordinates to include in the function call.

Updating the Radius, Angle, and Angular Velocity

The functions to update the radius, the current angle, and the angular velocity are simple.

```
void satellite::dist(float rad)
{
   radius=rad;
}
void satellite::angle(float ang)
{
   omega=ang;
}
void satellite::speed(float ws)
{
   wspeed=ws;
}
```

Moving Around

Finally, the function to move the satellite from one position to the next is as follows:

```
void satellite::move()
{
   omega=omega+wspeed;
   polarxy(radius,omega,xplanet,yplanet,x,y);
}
```

It is very simple, isn't it? All you do is to increment the current angle, and then compute the new cartesian coordinates. There is no need to invoke the `place(x,y)` function. The cartesian coordinates are directly updated with this function call.

The New Stars Program

Once the satellite class is available (`c6satell.h`), you can write a new version of the `c5stars.cpp` program (`c6sat.cpp`).

You can declare objects and initialize them as follows:

```
const float pi2=2*3.14159;
Stage universe;
Clock timer,sidereal;
satellite Sun,Earth,Moon;
Sun.resize(40);
Sun.origin(320,240);
Sun.scale(1.,-1.);
Earth.dist(120);
Earth.speed(pi2/36.5/30.);// In a 30th of a second
Earth.center(Sun);
Moon.resize(12);
Moon.dist(80);
Moon.color(3);
Earth.color(4);
Moon.speed(pi2/2.8/30.);  // In a 30th of a second
universe.insert(Earth);
universe.insert(Moon);
Sun.show();
```

The animation loop is simple.

```
for(;sidereal.time()<36.5;)
   {
     Earth.move();
     Moon.center(Earth);
     Moon.move();
     universe.show();
     timer.watch(.033);
     timer.reset();
     universe.erase();
   }
```

Why is the Moon told to center using Earth inside the loop? Couldn't this be done only once before the loop, as it was done with the Sun? No, it could not!

The problem is that Earth's coordinates are changing, and what we actually give the Moon are the new coordinates of Earth. This procedure is not needed for the Sun, because it does not move. Now do you see why it was useful to be able to change the center coordinates?

From Outer Space to Inner Space

What else can you do with satellites? Electrons are also satellites of the nucleus of an atom.

You can write a program to simulate electrons orbiting a nucleus just as easily as you wrote the previous program. If you want 10 electrons orbiting the nucleus, declare 10 satellites, initialize them, move them.… Wow—it may become an extensive program! Well, not really—wait until you learn about arrays!

Are You Experienced?

Now you can…

Choose appropriate classes to implement your applications

Evaluate possible improvements to your designs