PART I

GETTING READY

Part I will develop your basic skills in understanding and writing simple programs. By now, you should know how to use the C++ compiler as described in Part 0.

Computer programs are nothing but directions that tell the computer how to solve a problem. The simplest kinds of problems can be solved in a straight sequence of steps. These sequential programs do not check for any conditions, nor do they contain any repetitions. We will be dealing with this kind of program in Part I.

By the end of Part I, you will be able to understand, modify, and create simple programs; to communicate with the user; and to solve simple problems.

SKILL ONE

MAKING AND MODIFYING PROGRAMS

- Understanding program statements
- Sending messages to objects
- Understanding arguments
- Using athlete objects
- Adding comments

This skill is to help you understand how simple computer programs are written. We'll use a friendly "athlete" named Sal in several programs. You'll be able to display Sal on the screen, where he will perform some exercises and say a few words.

A Simple First Program

We start with a simple computer program that you can immediately run on your computer. Try this program as soon as possible because later you'll need to know how to use your computer to run other programs. To run this program, you use the project facility of your C++ compiler.

See Skill 0 for instructions on the appropriate way to run programs in your environment.

This program, `c_sal.cpp`, displays a picture on the screen. Here is the actual content of the program:

```
#include "franca.h"
athlete Sal;
void mainprog()
   {
       Sal.ready();
       Sal.say("Hi!");
   }
```

Let's look at what each line does.

#include ...

This line tells the computer to fetch other lines that I, the author, have prepared and include them in your program. These extra lines contain instructions for the computer to make your programs easier and more interesting. The computer knows these lines by the file name `franca.h`. As a matter of fact, this line does not belong to the C++ language. It instructs the compiler (more specifically a part of the compiler called the *preprocessor*) to locate these previously prepared program parts and pretend they were inserted at that point in your program.

It is possible to use pieces of programs that either you or somebody else developed in your programs. This is a real time saver! Several actions that we need to perform frequently may be provided with the compiler, bought from an independent vendor, or made available by colleagues. We then use the `#include` directive to incorporate them in our programs.

athlete Sal;

We are going to use one particular object to help you write your first programs. This object creates a picture in which one person performs fitness exercises. In `franca.h`, we explain to the computer that we will use one particular kind, or class, of object; we call this class `athlete`.

We named our athlete Sal. This line simply explains that Sal is an object of class `athlete`. Since objects of the class `athlete` have pictures in several positions, Sal will have these, too.

void mainprog()

This line begins to describe the main set of actions you want the computer to perform. These actions consist of the instructions enclosed within the opening and closing braces ({ }) shown on the next line and a few lines below.

In C++, braces delimit a set of actions.

Actually, in C++, the main set of actions is denoted by `main` or `winmain`. The name `mainprog` is used only with the software for this book. Later skills explain this in detail.

Using `mainprog` to name the main function is an educational trick. The actual `winmain` function is embedded in the supplied software. This software, which interfaces with Windows' graphic environment, has to be known by this name to the system. However, our programs will call the main function `mainprog`, so that you can explain what you want done. We will be naming our programs `mainprog` until we reach Part VIII.

Sal.ready();

This line says that we want Sal to be ready (ready means to assume the initial position for the exercise). To do this, we send a message (`ready`) to our object (`Sal`). The usual format for sending messages is the object name (`Sal`), a dot, and the message name (`ready`). The message name is followed by parentheses.

Sal.say("Hi!");

This is another message we send to Sal. We want Sal to say hi, and this is an easy way to do that. Will Sal actually speak? Well, not yet, but you will get his message! Notice that, in this case, something is inside the parentheses—what we want Sal to say. Enclose Sal's word in double quotes.

You can run this program right now if you like. Put your C++ compiler to work, and open the appropriate project. Examine the program and run it! Figure 1.1 shows the result.

> **RUNNING YOUR C++ PROGRAMS**
>
> If you have not read Skill 0 yet, please do so now. It explains how to run the programs you create in this book in your C++ compiler. Skill 0 shows you in detail how to use some of the most popular C++ compilers.

Sending Messages

Now that you know how to run a program, let's get started writing some. You will tell the computer to work with things called objects.

To get going in an easy and interesting way, I have provided some objects for your use. You already met the first one, Sal. Sal will be our companion during the first skills. Sal is an object that can be displayed in several positions:

ready
up
left
right

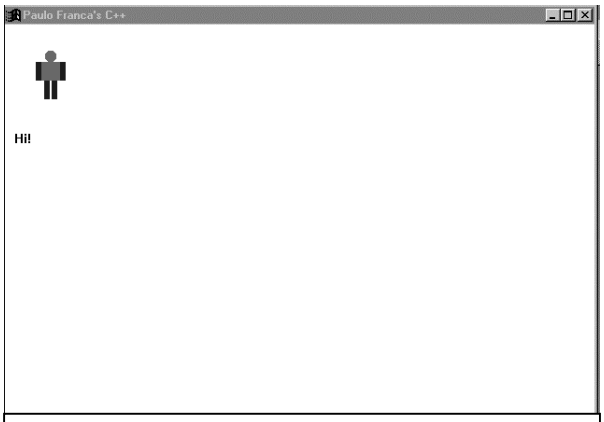


Fig 1.1 The result of c_sal.cpp

We have to send Sal the appropriate message so that he appears in the desired position. You may have already guessed that we have built our athlete so that he understands messages such as ready, up, left, and right and can respond to them by assuming the corresponding position.

In the generic programming literature, we say that we send a message to an object when we want that object to do something. In the more specific C++ literature, we say that we use one of that object's *member functions* when we want it to do something.
Both descriptions refer to the same thing. The term *message* may seem more appropriate at this stage, but you will get a better understanding of why to use member functions instead of messages when you learn about functions and classes in later skills.

In the first program, we simply instructed Sal to assume the ready position and to say hi. Before we go to the computer and send Sal some messages, however, let's look at a few programming basics.

Write One Statement per Line

Statements are instructions to your computer. Some statements are messages to objects. *Messages* are instructions that objects know how to obey. A computer (and its objects) can obey one instruction at a time. You can write more than one instruction left to right in the same line. However, computer programmers have long established that it is more convenient and understandable to place each instruction on a separate line.

You can only use messages that your objects understand.

In this case, we can use only messages such as ready, up, left, and right. If you tell Sal to go down, for example, he won't obey because he doesn't know how to do that yet!

The class `athlete` objects were designed to respond only to these specific messages (i.e., only these member functions were included in the class).

Figure 1.2 summarizes the actions that Sal can perform.

You must specify which object is supposed to obey your messages.

Some programs may deal with several objects. For example, you might have two athletes, Sal and Sally. If you simply say up, who do you want to move up? Sal? Sally? Both? You must precede the message with the object name, and you must place a dot between the object name and the message name, as in the following:

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

You must let the computer know which objects you are using and what class (what kind) of objects they are.

The C++ compiler differentiates between uppercase and lowercase letters. `Sal` is not the same as `sal`. This is called *case sensitivity*, and you must take it into consideration.

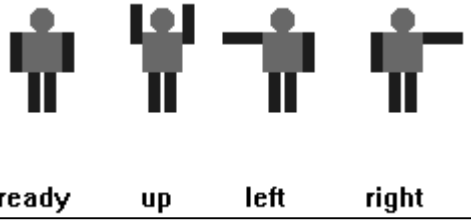


Fig 1.2 What an athlete can do

Be sure your commands and class names are the same throughout your program. Using *Sal* in one place and *sal* in another can cause errors.

Each instruction in your program must end with a semicolon (there are a few exceptions, such as the `include` and `void mainprog` instructions).

To send a message to an object, you use the object name, a dot, and the message followed by opening and closing parentheses (later we may include something inside the parentheses). For example:

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

tells the object `Sal` to position itself as ready.

Bugs?

You may have already noticed that any minor mistake (such as misspelling a name, missing a dot, and so) causes the computer either to send you an error message or to act improperly (if you have not seen this yet, do not despair—you will see it sooner than you expect). This is a great time to learn that the computer always does as it is told.

If the computer is not doing the right thing, it is because you told it to do the wrong thing! You may sometimes feel that the computer is crazy, but if you carefully examine your instructions, chances are you will find you did something wrong. These mistakes are called *bugs* in programming jargon.

The *athlete* Class

You can declare objects of class `athlete` and use them with your programs. All objects of this class have the following characteristics (summarized in Figure 1.2).

Each object occupies a square on the screen. The first object occupies the upper-left square, and each succeeding object you declare occupies a square to the right.

These objects have a shape that is drawn in response to the messages:

ready
up
left
right

In addition, athletes pretend to "say" something to you. To make this happen, send the message `say` to the athlete (enclose the message in parentheses and enclose the words you want spoken in double quotes).

Understanding Arguments

Arguments modify the effect of a message. For example, it is possible to alter the speed at which the athlete works by specifying how many seconds the athlete remains in the requested position. The `say` message also has an argument that specifies what you want the athlete to say.

Movements can be performed faster or slower. To modify the speed, all you have to do is to include the number of seconds as an argument inside the parentheses. For example, the following lines:

```
Sal.left(5);
Sal.up(8);
```

cause Sal to remain in the left position for 5 seconds and then switch to the up position and stay like that for 8 seconds.

The `ready`, `up`, `left`, and `right` arguments can be a number with a fractional part, for example:

```
Sal.left(0.5);
```

If you do not provide an argument, 1 second is assumed. If you use zero as an argument, the action is performed as fast as the computer can manage to do it. (You probably won't even be able to notice the movements!)

`Ready`, `up`, `left`, and `right` can use numeric arguments, that is, a number that specifies a length of time. On the other hand, `say` can use either a number or a sequence of characters, digits, and signs and will display them on the screen exactly as they appear inside the double quotes.

Please don't forget! If you save the modified program, you may lose the original! To preserve the old program, choose File ➢ Save As, and give the file a new name.

Why is it that when we want Sal to say something we enclose that "something" in quotes? Well, suppose you want Sal to say hello. Sending your message like this:

```
Sal.say(hello);
```

causes a problem. Hello is a valid name for an object, and the computer cannot tell whether you want to write the letters *h-e-l-l-o* on the screen or write the contents of the object `hello`. Therefore, the quotes tell the computer that you want to display exactly what is inside the quotes.

You must always use quotes ("") in C++ to denote text that will appear on the screen. C++ compilers differentiate between `(hello)` and `("hello").`

Keeping Your C++ Programs Nice and Neat

A good programmer is always concerned that others can understand what he or she has done. Never assume that all you have to do is have your program running. You or someone else may have to use and modify it. Worse still, you may have to understand and modify a program written by someone else! In any case, it is important that someone else can read and understand your program without too much trouble.

One of the ways you help someone else understand your programs is to use *comments*, explanatory lines that you insert in your code. These lines have nothing to do with how the program works, and the computer ignores them. They simply explain to someone else what you are doing in the program. (At a later time, they can also serve to remind you of what you were doing.)

You may feel reluctant to use comments in the programs you are working on right now because they are small and (hopefully) easy to understand, but don't be fooled! Your programs will grow in size and complexity. Start using comments now!

You add comments in C++ by preceding the text with two slashes (//). If you do this, all text to the end of the line is considered a comment.

For example, our first program could look like this after adding comments:

```
// *******************************************
//                          Program C_SAL.CPP
// This program shows an athlete on the screen.
// Programmed by Paulo Franca, 08/21/94
// Last revision on 08/21/94
// *******************************************
//

#include "franca.h"    // Use textbook programs
athlete Sal;           // Declare Sal to be an athlete
void mainprog()        // The program starts here
{
  Sal.ready();         // Tell Sal to appear in the
                       // Ready position.
  Sal.say("Hi!);       // Tell Sal to say "Hi!"
}                      // Program ends here.
```

You can write anything you like after the two slashes. If the text does not fit on one line; no problem. Use the next line, as I've done in the `Sal.ready()` statement. You can use the slashes at the beginning of the line or in any place you please. Just don't forget to use them!

Besides explanatory comments, it's always a good idea to include your name, the date you wrote the program, the date when you last changed it, and a brief description of what the program does. The description is helpful when you're later looking for a particular program.

Notice that between the comments:

`The program starts here`

and

`Program ends here`

all comments are indented to the right. This makes it easier for you to readily identify where actions start and finish. Use this same structure for program statements (notice that, inside the braces, all statements are indented).

Understanding Sequence

To execute a task, you must explain to the computer the steps involved. Of course, you must know how to accomplish the task to explain it.

An *algorithm* is a detailed, precise description of how to execute a task, and it can be expressed in any language. A *program* is an algorithm in a form that the computer can understand.

Here, we are going to explore only *sequential* algorithms, which involve a sequence of steps that are executed one after the other. In later skills, we'll look at how to include repetitions and decisions.

Taking One Step at a Time

If you can write the recipe for cooking Eggs Benedict or if you can write the directions to get to the downtown theater, there is no reason you can't explain to a computer how to solve a problem. Just remember a couple of things:

- You cannot explain what you don't know. You can't tell someone how to find the downtown theater if you don't know where it is. No matter what you want to explain to a person or to a computer, be sure you know what you are talking about.

- Nothing is obvious to a computer. A human is never as dumb as a computer. The computer has no judgment. If you don't explain all the details, the computer may get stuck.

Giving Sal Some Instructions

Let's start by telling Sal to perform some fitness exercises, for example:

```
ready
up
ready
```

Do you think you can write a program to do that? All you have to do is to send Sal the appropriate messages. Why don't you simply take the original program and include the new messages you want Sal to receive? Go on! Try it! Remember though:

- Write one instruction in each line.
- Don't forget the semicolon.
- Be sure that messages to Sal include his name, a dot, and the message followed by (). For example, `Sal.right()`, `Sal.left()`, etc.

If you do this exercise correctly, you should be able to see Sal move his arms up and then down.

Do you want to try another one? How about this:

```
ready
up
left
up
ready
up
right
up
ready
```

If you write these programs properly, you can see Sal exercising. If you want to try some other exercises before proceeding, be my guest! I want to make sure you understand that things happen because the computer looks at the instructions and executes them one at a time.

Try These for Fun

- Change the above program to make the fitness exercises go slower. Place a number, such as 5, inside the parentheses for the messages `ready`, `up`, `left`, and `right`.
- Change the program to make the exercise go faster. Place a zero inside the parentheses this time.
- Change the program `c_sal.cpp` so that Sal says "Sal" instead of "Hi!"
- Declare an object in addition to Sal, let's say Sally. Have each object say his or her name.
- Have Sal and then Sally perform the following fitness exercise:

```
ready
up
left
up
right
```

up
ready

- Have Sal say what he is doing (ready, up, left, up, right, up, done!) while he exercises.

Are You Experienced?

- Now you can…
- Identify program statements inside a C++ code list
- Send messages to objects of different types, including strings (text) such as “Hello!” and numbers
- Understand sequential algorithms
- Use athlete objects
- Use comments

SKILL

TWO

DISPLAYING INFORMATION TO THE USER

- Using Clock objects
- Using Box objects
- Using Robot objects

In this skill, you'll learn how to keep track of the time elapsed while running your programs and how to make the computer wait for a specific time. You do this with the objects of class `Clock`.

This skill also allows you to make the computer communicate with you or any other user. Often you will need to display some values or some words that reflect what your program is doing or its result. Box objects are convenient for this purpose. You can use a box to display a value or a message. Each box can also have a label that identifies its purpose.

To develop more interesting programs, you will learn how to use another class of objects, the `Robot` class. Objects of this class are useful for drawing simple shapes on the screen. (Later, we'll use them to illustrate more complex problems.)

Using *Clock* Objects

You declare objects of type `Clock` (notice the first character is uppercase) to keep track of time in your programs. Just like any other object used in the program, you have to declare an object by giving it an identifier:

```
Clock mywatch;       // mywatch is a Clock
```

After you declare this object, you can use it to do the following:

Keep track of the time elapsed since the object was declared

Wait for a specific time period before continuing the program

You can restart the clock at any time.

Messages to *Clock*

`Clock` objects can respond to the following messages:

```
wait(seconds);
time();
reset();
```

The `wait(seconds);` message instructs the computer to wait for the number of seconds specified as an argument inside the parentheses. This number can include a fractional part.

The `time();` message causes `Clock` to produce a result that is the time (in seconds) elapsed since `Clock` was initialized or created (you'll see how to use this later).

The `reset();` message resets Clock. Each clock is automatically reset (to zero) at declaration, so you don't need to reset your clock before using it.

Putting Words in Sal's Mouth with *Clock*

Suppose you want to use a clock to make Sal say, "Hello!" and, then after 3 seconds, say, "How are you?" All you have to do is the following:

1. Send a message for Sal to say "Hello!"
2. Send a message to your `Clock` object to wait 3 seconds.
3. Send a message for Sal to say "How are you?"

You must, however, declare an object of class `athlete` (`Sal`) and declare an object of class `Clock` so that you can use it. Also, you must send a message for Sal to show himself (`ready`); otherwise, he will be invisible!

In the program below, `C1CLOCK.CPP`, an object named `mywatch` functions as the clock.

```
// This program illustrates the use of Clocks
// Programmed by: Paulo Franca, 10/19/94
#include "franca.h"                        C1CLOCK.CPP
void mainprog()
{
  Clock mywatch;
  athlete Sal;
  Sal.ready();              // Show Sal...
  Sal.say("Hello!");        // Say Hello!
  mywatch.wait(3);          // wait 3 second before
                            // doing what is next
  Sal.say("How are you?"); // now say "how..."
  mywatch.wait(1);          // wait 1 second...
}
```

The actual arguments to the `Clock` object are numbers that can contain a fractional part. The time returned by the `time()` message also has a fractional part. You can, therefore, specify a wait of a half second by using a statement such as the following:

```
mywatch.wait(0.5);
```

The `mywatch` object was automatically reset to zero when the program started execution. We could, at any moment, display the elapsed time in seconds by including a statement such as the following:

```
Sal.say(mywatch.time());
```

We could see the actual elapsed time since the beginning of the program, because `mywatch.time()` causes `Clock` to return the value of the elapsed time. So far, the only way we know to show a value is to have the athlete "say" the value.

Getting Started

When you have to develop a new program, you sometimes feel stuck and stare at the computer screen without a hint of how to get started. This is a common beginner's symptom. I suggest you consider developing the habit of writing down some of the actions that are needed in plain English. If you like, use the computer, and enter those actions as comments. As you do so, your mind will get organized, and you'll start to understand the problem.

For example, you want to display one athlete (Sal) who says,"Sally! Where are you?" Then, after 5 seconds, you want to display another athlete (Sally) who says, "Here I am!"

An athlete can say only short words, because there isn't much space provided in the display box. You can, however, have the athlete say a longer sentence by breaking it into smaller pieces. Thus, instead of Sal's saying "Sally! Where are you?" all at once, he can say this sentence a few words at a time, with a 2-second interval between words.

If you follow my suggestion about writing the actions as comments, you might have the following:

```
// declare two athletes
// Show Sal on the screen
// have Sal say the words "Sally! Where are you ? "
//      waiting 2 seconds for each word.
// wait 5 seconds
// Show Sally
// have Sally say: "Here I am"
//      waiting 2 seconds for each word.
```

These comments actually describe the algorithm of the program. You might look at them and feel ready to write the program. If not, change them or provide better explanations until you can see the program completely explained.

You might not like the comment that tells Sal to say a sequence of words, waiting 2 seconds between each word. Perhaps you want to explain this better. No problem—simply change that comment to something like the following:

```
// Have Sal say "Sally!"
// wait 2 seconds
// have Sal say "Where"
// wait 2 seconds
// have sal say "are you?"
```

Insert these comments in the appropriate place, and the complete algorithm will be as follows:

```
// declare two athletes
// Show Sal on the screen
//        Have Sal say "Sally!"
//        wait 2 seconds
//        have Sal say "Where"
//        wait 2 seconds
//        have sal say "are you?"
// wait 5 seconds
// Show Sally
// have Sally say: "Here I am"
//      waiting 2 seconds for each word.
```

The *indention* makes it clear that the purpose of those actions is to perform the complete action in the line above (`Sal say the words...`). You might want to provide similar comments to explain how Sally will say her part, or you can leave the algorithm as it stands.

Adding Statements to Your Comments

Now that you have a good description of your program, you can leave each comment exactly where it is and start to include the actual C++ statements. For example, just below the following line:

```
// declare two athletes
```

include something like this:

```
athlete Sal, Sally;
```

Add all the statements, and your program is ready. Try to adopt the practice of outlining the steps of your program as comments and then adding the statements. It can be especially useful if you don't know how to get started.

As an exercise, try to complete this program and run it.

Using *Box* Objects

Suppose in one of the above programs you want to display how long it takes for the program to complete. You can easily get this information from the `Clock` object. But where do you display this value?

The only option we have covered so far is to have Sal "say" the value of the time. If you want to display this information in a different manner, you need a different object.

`Box` objects are similar to `athlete` objects. Once you declare a `Box` object (notice the uppercase B), you can use it to "say" whatever you like, just as you did with an `athlete` object. `Box` objects differ from `athlete` objects in the following two ways:

No athlete is associated with them.

The `Box` object can have a label.

Boxes are automatically located one below the other on the right side of the screen.

For example, to display the time, you can declare a `Box` object as follows:

```
Box display ("Time:");
```

At any point, you can now use this box to display the value of the time, which is kept by an existing clock, `mywatch`:

```
display.say(mywatch.time());
```

In this case, the word *Time* is the label of the box, as stated in the declaration. The time value is shown as the content of the box.

If you declare more than one box, the next boxes appear one below the other. In Skill 14, you will learn how to move boxes to other locations.

The Great Escape—Part 1

You use `Robot` objects to simulate a situation in which you and your master are lost in a maze and explore the alternatives.

Robot Objects

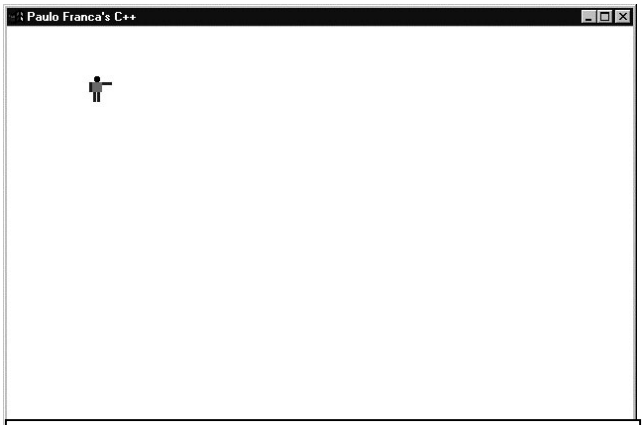


Fig 2.1 A Robot object

`Robot` objects consist of a robot inside a room that may or may not have walls blocking the way. Standard `Robot` objects are placed in an empty room in the upper-left corner of the screen. It is a good idea to force the robot to face a specific direction, say east, when you start. Figure 2.1 illustrates a robot in this situation.

To use a `Robot` object in your program, you must declare one (and only one) `Robot` object. Throughout this book, all the examples declare a `Robot` object named Tracer. This will remind you of the robot you met in the maze story.

The program `c1robot1.cpp`, listed below, was used to generate the screen in Figure 2.1.

```
#include "franca.h"
// c1robot1.cpp
Robot Tracer;
void mainprog()
   {
      Tracer.face(3);
      Tracer.step();
   }
```

What Can the Robot Do?

Tracer, as well as any other `Robot` object you use in your programs, knows how to respond to the following messages.

step()	Takes one step ahead. Optionally, you can specify a number of steps inside the parentheses. The robot does not check for a wall. Each step takes a standard amount of time, as set by `timescale` (see below). The default time is 1 second.
left()	Turns 90 degrees to the left.
right()	Turns 90 degrees to the right.
seewall()	Checks for a wall ahead (next square). (We'll use this message in Skill 8.) This message returns a one if there is a wall, and zero otherwise.
seenowall()	Checks for no wall ahead (next square). (We'll use this message in Skill 8.) This message returns a one if there is no wall ahead, and zero otherwise.
face(integer)	Turns the robot to face the specified direction. Direction is an integer (0, 1, 2, or 3), specifying north, west, south, or east. If any other value is given, the remainder of the division by 4 is used.
mark()	Marks the next square in the map using a given color. If no value is specified, two is used (green). Table 2.1 shows the colors and their corresponding values.
say("message")	Displays the "message" on the screen, as if the robot had said it. The message can be either an integer (in which case, omit the quotes) or a few words (enclosed in quotes). Avoid exceeding 12 characters in a string.
timescale(value)	Changes the `timescale`. You don't need to use this unless you want to change the default value, in which each step takes 0.1 seconds. For example, if you want to slow down Tracer so that she spends 0.5 seconds in each step, you can do this as follows: Tracer.timescale(0.5)

Table 2.1: The Colors and Their Values

COLOR	ITS VALUE
White	0
Red	1
Green	2
Blue	3
Light Blue	4
Pink	5
Yellow	6
Black	7

The C++ syntax for handling all objects is exactly the same whether you are dealing with athletes, clocks, or robots. To send a message to an object, you use the object name, a dot, and then the message followed by parentheses. Therefore, as shown in the `c1robot1.cpp` program, to have Tracer step ahead, you use a statement such as the following:

```
Tracer.step();
```

Tracer appears on the screen with her arm pointing in the direction in which she just stepped. In the following examples, Tracer will move in an empty room, so you don't need to check for her bumping into the walls. Later, you will learn how to check for walls on the way and how to avoid crashing.

Moving Your Robot

Now, let's modify `c1robot1.cpp` and experiment. To start, have her step four times. You can do this by repeating these steps:

```
Tracer.step();
Tracer.step();
Tracer.step();
Tracer.step();
```

or by indicating the number of steps inside the parentheses:

```
Tracer.step(4);
```

This will do for the time being. Later, you will learn how to tell the computer to repeat a sequence of statements.

Marking Tracer's Way

Now, let's paint the squares as Tracer moves so that you can clearly see her path. A robot can mark the square that is immediately in front in a given color. Can you modify the program so that Tracer marks the squares as she goes?

Remember, Tracer cannot mark the square she is stepping in. Only the one immediately ahead of her.

If we don't care about the first square, the following program will do:

```
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
```

With this program, Tracer marks and steps four times. The initial square remains blank.

Turning Tracer

A robot can turn to the left or to the right and can face any of four directions. Directions are represented by an integer number, as shown in Table 2.2.

Table 2.2: Directions and Their Values

DIRECTION	ITS VALUE
North, facing the top of the screen	0
West, facing the left side of the screen	1
South, facing the bottom of the screen	2
East, facing the right side of the screen	3

Suppose, for example, that after Tracer steps and marks four squares, we want her to turn back to the original position.

How can we do that? Well, you have to make Tracer turn around and then step four times. This should bring her back to the original square. How do you turn back? Either you turn to one of the sides twice, or, in this case, since Tracer was facing east, have her face west:

```
Tracer.face(1);
```

Putting Words in Tracer's Mouth

At any point, the robot can inform you of what's happening in the program. As can athletes, robots can say something. For example:

```
Tracer.say("Done!");
```

By the way, did you notice that robots and athletes look alike? Hmm, maybe they are related! Who knows?

Now you can modify your program so that Tracer says "going" as she steps and marks the four squares ahead. When Tracer turns back, she says "coming back" and then steps back to the original square.

Helping Your Master

Imagine that you, your master Ram, and Tracer are lost in a maze built with rectangular squares. You are now in a position to start helping Ram explore the maze. As an introductory exercise, Ram has suggested that the robot be instructed to walk, describing a square of size 3. As you know, Tracer has to step 3 times, turn to the right, step 3 times, and so on.

This was the algorithm:

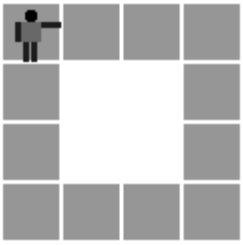
step
step
step
turn right
step
step
step
turn right
step
step
step
turn right
step
step
step
turn right

We can easily transcribe this into a C++ program:

```
// Program c1robot2.cpp
#include "franca.h"  //c1robot2.cpp
Robot Tracer;
void mainprog()
{
     Tracer.face(3);
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step(); // done with one side
     Tracer.right();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step(); // done with one side

     Tracer.right();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step(); // done with one side
     Tracer.right();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step();
     Tracer.mark();
     Tracer.step(); // done with last side.
     Tracer.right();
}
```

I'm sure you're upset about writing all these steps. Hold on until the next skill! Figure 2.2 illustrates the result of running this program.



Is there anything wrong with this program? Of course, there is! Why does the square measure 4 steps instead of 3? Well, it turns out that the program is wrong! In this program, the original square was not counted as part of the square. Therefore, stepping 3 times for each side actually results in a square of size 4, as shown in Figure 2.2. To draw a square of size 3, step 2 more times beyond the current square.

Fig 2.2 The Robot's journey

Are You Experienced?

Now you can…

Keep track of time using a Clock object

Display data using a Box object

Use a robot

SKILL

THREE

SOLVING PROBLEMS

- Devising solutions to simple problems
- Understanding syntax and rules

The purpose of this skill is to start developing your ability to work on problem solving. As you work through this book, you will no doubt improve on this skill, as you will throughout your programming life. I cannot teach you how to devise a solution to any problem. I can give you some hints on how to do that yourself.

At the end of this skill, you will find a more formal discussion of the rules for writing programs. I purposely omitted most of the rules so that you could concentrate on the concepts: what is the purpose of a statement, why do we use it, and so on. However, rules must be very strict for the compiler, and, therefore, we'll now take a look at them.

Facing and Solving Problems

The most difficult part is to overcome the fear of starting. Write down any explanation that comes to your mind. If your explanation is wrong, no problem. It is easier to correct an explanation once it is written. If the explanation is vague, just go on and add more detail to it.

Write Anything

Never trap yourself by staring at a blank piece of paper (or at a blank screen, for that matter). You will never get anywhere that way.

Once you have written some explanation of how to solve your problem, read it a few times. Is it correct? It's all right if is isn't precise. For example, the following explanation:

Draw a square of size 3.

is correct. That is what you want to do. However, it is not precise. It won't do for the computer. You need to explain what you mean by drawing a square of size 3!

Break Down the Problem

You must go on explaining your problem until the computer is able to understand how to solve it. The general idea is to identify smaller problems that you already know how to solve. This will get you closer to the solution. Try to break the problem down into easier ones, but, if possible, into problems for which you already have a solution. For example, to draw a square, you might think of drawing a line at a time:

```
draw a line
turn right
draw a line
turn right
draw a line
```

turn right
draw a line
turn right

It would be great if your computer already knew how to draw a line. But that is no problem. You can now explain how to draw a line:

mark
step
mark
step

This explanation gives you the proposed algorithm.

Reuse Simpler Problems

Finally, let me stress that, while breaking down your problem into simpler ones, try to identify simpler problems that you can reuse. What do I mean by reuse? If you can fit a solution that you already have, you are reusing that solution. The more you reuse, the more time you save. Sometimes, though, you don't have a reusable piece of software available. What do you do?

Well, in that case, you will have to build it yourself. Nonetheless, whenever you build a piece of software, try to evaluate how it could be reused in the future. Try to build your software, or , most likely, your functions (as you will see in the next skill), in such a way that they can be reused without any change.

Why Are Computers So Dumb?

Why are computers puzzled by so little? This next exercise introduces some intentional mistakes in the program. It is designed to get you used to some of the most common error messages. Notice that, in many cases, the error message does not explain the problem well enough.

Take the program `c_sal.cpp` and try the following one at a time:

- Remove the semicolon at the end of a statement:

```
Sal.ready()
```

- Remove one opening parenthesis:

```
Sal.ready);
```

- Remove the opening braces.
- Remove the `#include` line.
- Remove the following line:

```
Sal.say(Hi!);
```

Most likely, you will receive an error message indicating that the compiler got confused and could not understand what you wanted. Try to understand why these messages show up and what caused the confusion. For example, removing a semicolon causes the compiler to think that the statement has not ended, and so the computer continues in the following line. Now, since

```
Sal.ready() Sal.say("Hi!");
```

does not represent anything the computer can recognize, the compiler does not know what to do. This may seem like a small mistake that you can easily fix, but the computer has to work with strict rules and is unable to find out what was wrong.

Most of the above cases fall into the category of *syntax errors*, which are violations of simple rules for constructing statements. For example, for each opening parenthesis, there must be a closing parenthesis; all object names must be declared; and so on.

The last error in the list, however, does not generate an error message. You may be able to run your program, but the program will not be correct because "Hi!" will not be displayed under the picture. In this case, the compiler could not find anything wrong. The syntax rules were

respected. Nevertheless, the program is not correct because it does not do what you intended in the original version.

Keep in mind that the computer always does what it is told to do. If something is not done correctly, chances are you did not issue the correct instructions. Unfortunately, because computers have a simple "mind," the rules have to be strictly observed. A minor mistake can cause a completely different action to take place!

Rules and Regulations

Thus far, I have loosely explained the rules for constructing programs. I was more interested in making sure you understand the purpose of the programming elements rather than the actual rules to use them. However, the rules are important, and I'll summarize them in this section. Hopefully, it will be easier for you to understand the rules after you are familiar with the purpose of each programming construct.

Identifier Names

To identify objects you use in a program, you designate them by an *identifier*, which is simply a name for the object. You can choose the identifier as you wish, subject to the following rules:

- Begin with a letter or an underscore (_).
- Use letters, digits (0 through 9), and an underscore.
- Use a maximum of 32 characters (you can use more, but the compiler will look at only the first 32).
- Do not use blank spaces.
- Do not use a keyword as an identifier (keywords are explained below).

It is a good idea to use names that remind you of the purpose of the object you are naming.


Long names may bore you after a while.

The following are not valid identifiers:

- `2waystop` (It starts with a digit.)
- `my number` (It contains a blank.)
- `this-number` (The minus sign is not valid.)
- `"Sal"` (The quotes are not valid.)

On the other hand, the following are valid:

- twowaystop
- mynumber
- thisnumber
- Sal

Keywords

Keywords have a special meaning for the compiler. You might have noticed that we have used a few of them, `void`, for example. You cannot use keywords to name your objects. As you

go through this book, you will be learning keywords. Most compilers have a *syntax highlight* feature. All keywords are displayed in boldface as soon as you type them.

Types and Classes

Types and classes serve similar purposes, and, for the time being, we will use the two terms interchangeably. I'll explain the technical difference in a later chapter.

Types or classes denote the general characteristics of a group of objects. As you will see shortly, there is a type of data designed to contain and operate with integer numbers. This is the type `int`, which contains a number without a decimal part. You can perform some arithmetic operations on an `int`. You also learned how to deal with other kinds of data, athletes, for example. Athletes can be set to ready, up, left, and right, and they can say something. All data of the same type have similar characteristics. All athletes know exactly the same tricks.

It is important that you tell the compiler the class of each object that you are using so that the compiler can verify that you are doing the correct operations with that object.

You can also create your own classes. For example, `athlete` is a class I created for your use. In later skills, you will see how you can create your own classes, either using an existing class such as `athlete` or starting from scratch.

Declarations

The declaration simply consists of the class of the object, followed by the identifier. The declaration has the following syntax:

```
<<class name>> << object name>> ;
```

The `class name` entry is a valid class name existing in your program (`athlete` and `Clock` are defined in `franca.h`), and `object name` is an identifier of your choice that denotes the object you want to use. You can declare more than one object by separating their identifiers with commas, as follows:

```
athlete Sal, Sally;
```

You can use the identifier only after it has been declared, and you can declare objects and variables anywhere in your program. Here is an example:

```
Clock mywatch      ;  // mywatch is a Clock.
int number_of_times;  // number_of_times is an integer
```

Messages

You can instruct the objects you use in your program to perform certain actions. To do so, your program sends a "message" to the object. Objects can respond only to messages to which they are prepared to respond.

To send a message to an object, you must include the object name (identifier), a dot, and the message name followed by parentheses. If the message requires arguments, enclose them in the parentheses. Here is the syntax:

```
<<object name>> . <<message>>  ();
```

The following example sends the message `up` to the object `Sal`:

```
Sal.up();
```

The following example sends the message `up` to the object `Sal` with an argument of `2`:

```
Sal.up(2);
```

The argument indicates that you want to execute the next instruction after 2 seconds have elapsed.

Technically speaking, C++ uses the term *member function* instead of the term *message*. We'll look at member functions in Skill 16.

Comments

You include comments in your program to make it easier for you or someone else to understand. The computer does not use comments. You can include comments in C++ programs in two ways:

- Enter two slashes (//) and then your text.
- Enter `/*`, type your text, and then finish with `*/`.

The latter method allows you to include several lines as a comment:

```
/* This is
a comment that
occupies 3 lines */
```

It also allows you to include comment in the middle of a statement:

```
int /* this is a comment */ lapcount;
```

Try These for Fun

Let's look at some examples.

Find the Error

The following is a "wrong" copy of our first program `c_sal.cpp`:

```
include "franca.h"
athlete Sal;
void mainprog()
{
sal.ready();
Sal.say(Hi);
}
```

This copy has three mistakes. Compare the program with the original and locate the mistakes. Then, try to run the wrong program and notice the error messages.

Write a Couple of Programs

First, write a program that displays three athletes and their names. Declare three athletes (with names of your choice), and send messages to each one to get "ready" and to "say" their name.

Second, write a program that displays a picture such as that in Figure 3.1 (Sal and his shapes). You need to declare four athletes and send a different message to each.

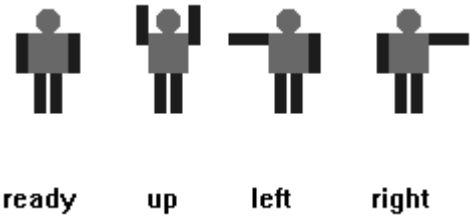


Fig 3.1 Sal and his shapes

Find Some More Mistakes

The following is another "wrong" copy of `c_sal.cpp`:

```
#include "franca.h"
void mainprog()
{
ready;
Sal.say("Hi!");
}
```

This program also has three mistakes. Locate the mistakes, and run the program to see the error messages.

Find the Valid Identifiers

Indicate which of the identifiers below are valid:

2ndTime
apollo-13
apollo13
BigTime
my friend
my.friend
my_friend
void

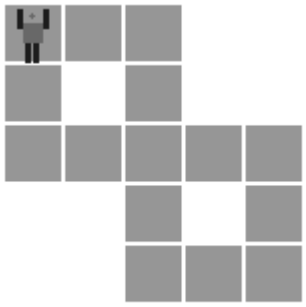


Fig 3.2 Your Robot is an artist

Draw a Picture

Write a program that instructs your robot to produce a drawing such as the one in Figure 3.2.

You can draw this picture in a couple of ways:

- Draw a square, move to the lower-right position of this square, and then draw the other square.
- Draw all the lines in a continuous movement.

Each move takes one second. Can you figure out which solution is the fastest? If the robot knew how to draw a square, which solution would be easier?

Find Some More Identifiers

Examine the following program. Which identifiers are invalid? Which identifiers were not declared?

```
void mainprog()
{
  athlete bill, john doe, ann3;
  Clock 3days, my-clock, O'Hara,other;
  Box show;
  show.say(thetime);
  bill.say(other.time());
  Marie.ready();
  Robert.up();
}
```

Are You Experienced?

Now you can…

Devise solutions to simple problems

Understand syntax and rules