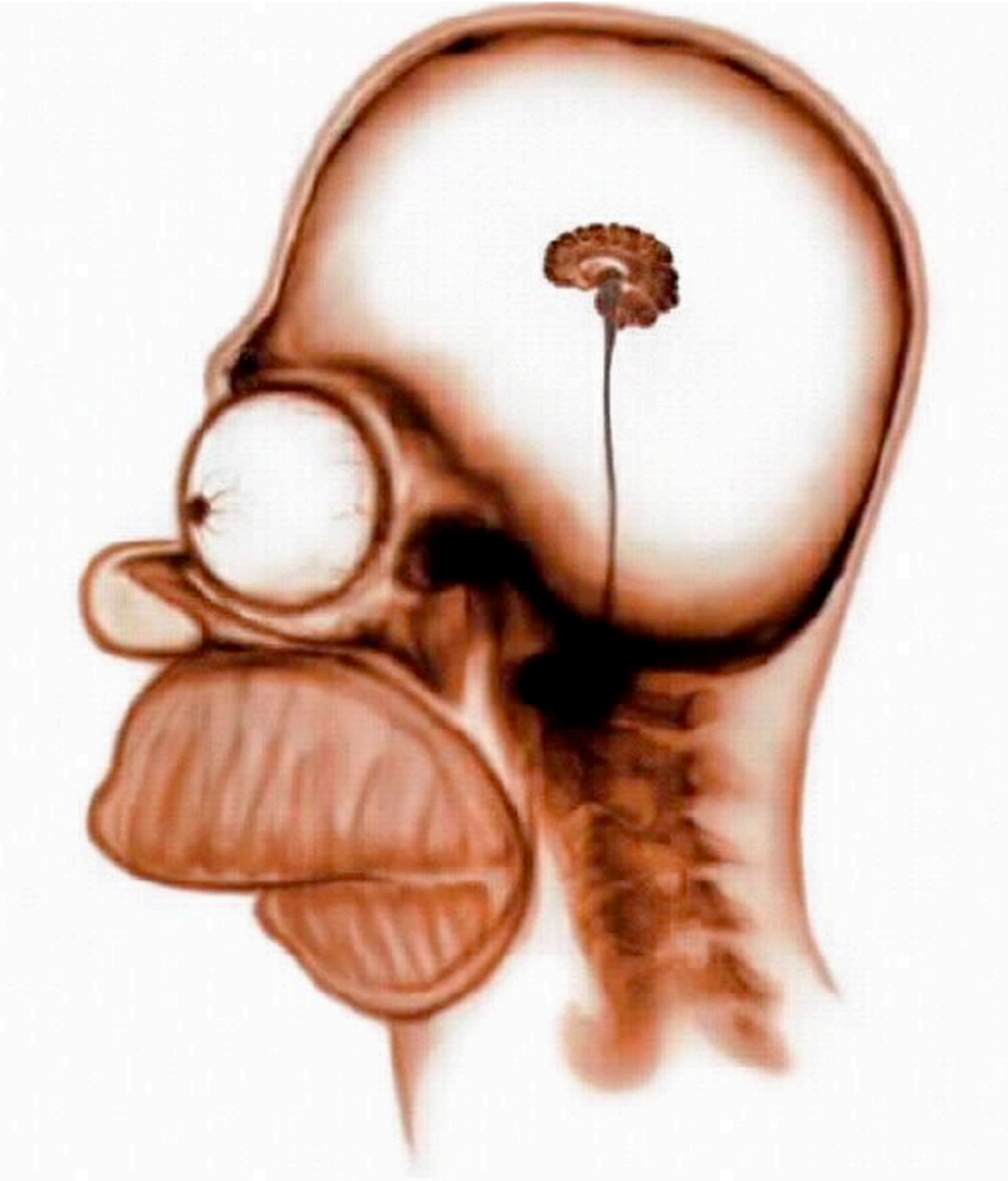

3

Building Robot Brains

What a splendid head, yet no brain.
Aesop (620 BC-560 BC)

Opposite page: Home Simpson's Brain
Photo courtesy of The Simpson's Trivia (www.simpsonstrivia.com)

If you think of your robot as a creature that acts in the world, then by programming it, you are essentially building the creature's brain. The power of computers lies in the fact that the same computer or the robot can be supplied a different program or brain to make it behave like a different creature. For example, a program like Firefox or Explorer makes your computer behave like a web browser. But switching to your Media Player, the computer behaves as a DVD or a CD player. Similarly, your robot will behave differently depending upon the instructions in the program that you have requested to run on it. In this chapter we will learn about the structure of C++ programs and how you can organize different robot behaviors as programs.

The world of robots and computers, as you have seen so far is intricately connected. You have been using a computer to connect to your robot and then controlling it by giving it commands. Most of the commands you have used so far come from the Myro library which is specially written for easily controlling robots. The programming language we are using to do the robot programming is C++. C++ is a general purpose programming language. By that we mean that one can use C++ to write software to control the computer or another device like a robot through that computer. Thus, by learning to write robot programs you are also learning how to program computers. Our journey into the world of robots is therefore intricately tied up with the world of computers and computing. We will continue to interweave concepts related to robots and computers throughout this journey. In this chapter, we will learn more about robot and computer programs and their structure.

Basic Structure of a Robot Brain

The basic structure of a C++ program (or a robot brain) is shown below:

```
int main() {
   <do something>
   <do something>
   ...
}
```

This is essentially the same as defining a new function. In fact, in C++ the *main program*, where execution begins, is always a function called `main`. In general, the structure of your robot programs will be as shown below (we have provided line numbers so we can refer to them):

```
Line 1: #include "Myro.h"
Line 2: <any other includes>

Line 3: <function definitions>
Line 4: int main() {
Line 5:     connect();
Line 6:     <do something>
Line 7:     <do something>
Line 8:     ...

Line 9:     disconnect();
Line 10: } // end of main program
```

Every robot brain program will begin with the first line (`Line 1`). This, as you have already seen, includes the Myro library and other declarations for communicating with the robot. In case you are using any other libraries, you will then include them (this is shown in `Line 2`). This is followed by the definitions of functions (`Line 3`), and then the definition of the function `main`. The first line of `main` (`Line 5`) includes the commands to connect to the robot. Finally, the last line of the function body (`Line 9`) includes the commands to disconnect from it. When you run this program it will begin execution on `Line 5` and finish after executing `Line 9`. In order to illustrate this, let us write a robot program that makes it do a short dance using the `yoyo` and `wiggle` movements defined in the last chapter.

```
/*
 * File: dance.cpp
 * Purpose: A simple dance routine
 */

// First include standard declarations

#include "Myro.h"
```

```
// Define the new functions...

void yoyo(double speed, double waitTime) {
       robot.forward(speed, waitTime);
       robot.backward(speed, waitTime);
}

void wiggle(double speed, double waitTime) {
       robot.motors(-speed, speed);
       wait(waitTime);
       robot.motors(speed, -speed);
       wait(waitTime);
       robot.stop();
}

// The main dance program
int main() {
       connect();
       cout << "Running the dance routine..." << endl;
       yoyo(0.5, 0.5);
       wiggle(0.5, 0.5);
       yoyo(1, 1);
       wiggle(1, 1);
       cout << "...Done" << endl;
       disconnect();
}
```

We have used a new C++ command in the definition of the `main` function: the `cout <<` command. This command will print out the text enclosed in double quotes (`"`) when you run the program. This is followed by `<< endl`, which ends the output line, so that subsequent output goes on a new line. This program is not much different from the `dance` function defined in the previous chapter except we are using a spin motion to wiggle. However, instead of naming the function `dance` we are calling it `main`. This is necessary in C++ to identify the main program in a program file.

Do This: In order to run this program on the robot, you can start your program editor or IDE, enter the program in it, save it as a file (`dance.cpp`) and then compile it according to the procedure on your computer system.

When you run the program you will notice that the robot carries out the dance routine specified in the `main` program. Also notice the two messages printed in your computer's window. These are the results of the output command (`cout <<`). This is a very useful command in C++ and can be used to output essentially anything you ask it to. While you are in this session, go ahead and change the output command to the following:

```
speak("Running the dance routine");
```

`speak` is a Myro command that enables speech output from your computer. Go ahead and change the other output command also to the speak command and try your program. Once done, enter some other speak commands in your program. For example:

```
speak("Dude! Pardon me, would you have any Grey Poupon?");
```

The speech facility is built into most computers these days. Later we will see how you can find out what other voices are available and also how to change to them.

Speaking C++

We have launched you into the world of computers and robots without really giving you a formal introduction to the C++ language. In this section, we provide more details about the language. What you know about C++ so far is that it is needed to control the robot. The robot commands you type are integrated into C++ by way of the Myro library. C++ comes with several other useful libraries or "classes" that we will try and learn in this course. If you need to access the commands provided by a library, all you have to do is include them.

The libraries themselves are largely made up of sets of functions (they can contain other entities but more on that later). Functions provide the basic building blocks for any program. Typically, a programming language (and C++ is no exception) includes a set of pre-defined functions and a mechanism

for defining additional functions. In the case of C++, it is the function definition construct. You have already seen several examples of function definitions and indeed have written some of your own by now. When defining a new function, you have to give the new function a *name*. Names are a critical component of programming and C++ has rules about what forms a name.

What's in a name?

A name in C++ must begin with either an alphabetic letter (`a-z` or `A-Z`) or the underscore (i.e. _) and can be followed by any sequence of letters, digits, or underscore letters. For example,

```
iRobot
myRobot
jitterBug
jitterBug2
my2cents
my_2_cents
```

are all examples of valid C++ names. Additionally, another important part of the syntax of names is that C++ is *case sensitive*. That is the names `myRobot` and `MyRobot` and `myrobot` are distinct names as far as C++ is concerned. Once you name something a particular way, you have to consistently use that exact case and spelling from then on. Well, so much about the syntax of names, the bigger question you may be asking is *what kinds of things can (or should) be named?'*

So far, you have seen that names can be used to represent functions. That is, what a robot does each time you use a function name (like `yoyo`) is specified in the definition of that function. Thus, by giving functions a name you have a way of defining new functions. Names can also be used to represent other things in a program. For instance, you may want to represent a quantity, like speed or time by a name. In fact, you did so in defining the function `yoyo`, which is also shown below:

```
void yoyo(double speed, double waitTime) {
      robot.forward(speed, waitTime);
      robot.backward(speed, waitTime);
}
```

Functions can take parameters that help customize what they do. In the above example, you can issue the following two commands:

```
yoyo(0.8, 2.5);
yoyo(0.3, 1.5);
```

The first command is asking to perform the `yoyo` behavior at speed 0.8 for 2.5 seconds where as the second one is specifying 0.3 and 1.5 for speed and time, respectively. Thus, by parameterizing the function with those two values, you are able to produce similar but varying outcomes. This idea is similar to the idea of mathematical functions: *sine(x)* for example, computes the sine of whatever value you supply for *x*. However, there has to be a way of defining the function in the first place that makes it independent of specific parameter values. That is where names come in. In the definition of the function `yoyo` you have named two parameters (the order you list them is important): `speed` and `waitTime`. By declaring each to be a `double` parameter, you have said that they are expected to be fractional quantities. (All C++ parameters must have a declared *type*, such as `double`, which specifies the allowed values of the parameter.) Then you have used those names to specify the behavior that makes up that function. That is the commands `forward`, and `backward` use the names `speed` and `waitTime` to specify whatever the speed and wait times are included in the function invocation. Thus, the names `speed` and `waitTime` represent or designate specific values in this C++ program.

Names in C++ can represent functions as well as values. What names you use is entirely up to you. It is a good idea to pick names that are easy to read, type, and also appropriately designate the entity they represent. What name you pick to designate a function or value in your program is very important, for you. For example, it would make sense if you named a function `turnRight` so that when invoked, the robot turned right. It would not make any sense if the

robot actually turned left instead, or worse yet, did the equivalent of the yoyo dance. But maintaining this kind of semantic consistency is entirely up to you.

Values

In the last section we saw that names can designate functions as well as values. While the importance of naming functions may be obvious to you by now, designating values by names is an even more important feature of programming. By naming values, we can create names that represent specific values, like the speed of a robot, or the average high temperature in the month of December on top of the Materhorn in Switzerland, or the current value of the Dow Jones Stock Index, or the name of your robot, etc. Names that designate values are also called *variables*. C++ provides a simple mechanism for defining variables, specifying their types and giving them initial values:

```
double speed = 0.75;
double aveHighTemp = 37.0;
double DowIndex = 12548.30;
string myFavoriteRobot = "C3PO";
```

These examples illustrate two of the many kinds of values in C++, namely *numbers* and *strings* (anything enclosed in double-quotes, `"`). The above are examples of *variable definitions* in C++. This is the basic syntax of a variable definition:

```
<type> <variable name> = <expression>;
```

Variables are so called because their values can be changed, which is accomplished with an assignment statement. For example,

```
speed = 0.9;
```

changes the value of `speed` to 0.9. (We do not mention the variable's type, `double`, because it was already declared in the definition of `speed`. In C++

the type of a variable cannot be changed once it is declared.) The exact syntax of an assignment statement is given below:

```
<variable name> = <expression>;
```

You should read the above statement as: *Let the variable named by* `<variable name>` *be assigned the value that is the result of calculating the expression* `<expression>`. So what is an `<expression>`? Here are some examples:

```
5
5 + 3
3 * 4
3.2 + 4.7
10 / 2
```

The simplest expression you can type is a number (as shown above). A number evaluates to itself. That is, a `5` is a `5`, as it should be! And `5 + 3` is `8`. As you can see, addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) can be used on numbers to form expressions that involve numbers.

You may have also noticed that numbers can be written as whole numbers (3, 5, 10, 1655673, etc) or with decimal points (3.2, 0.5, etc) in them. C++ (and most computer languages) distinguishes between them. Whole numbers are called *integers* and those with decimal points in them are called *floating point* numbers. In C++ the most commonly used floating point numbers are called `double`s, as you have seen. While the arithmetic operations are defined on both kinds of numbers, there are some differences you should be aware of. Look at the outputs produced by the following examples (suitably embedded in a main program):

```
cout << 10.0/3.0 << endl;
```

```
3.3333333333333335
```

```
cout << 10/3 << endl;
```

```
3
```

```
cout << 1/2 << endl;
```

```
0
```

```
cout << 1.0/2 << endl;
```

```
0.5
```

When you divide a floating point number by another floating point number, you get a floating point result. However, when you divide an integer by another integer, you get an integer result. Thus, in the examples above, you get the result 3.3333333333333335 when you divide 10.0 by 3.0, but you get 3 when you divide 10 by 3. Knowing this, the result of dividing 1 by 2 (see above) is zero (0) should not be surprising. That is, while the division operation looks the same (`/`), it treats integers differently than floating point values. However, if at least one of the numbers in an arithmetic operation is a floating point number, C++ will give you a floating point result (see last example above). You should keep this in mind. More on numbers later, before we get back to robots, let us quickly introduce you to strings.

Computers came to be called so because they excelled in doing calculations. However, these days, computers are capable of manipulating any kind of entity: text, images, sounds, etc. Text is made of letters or *characters* and strings are simply sequences of characters. C++ requires that strings be written enclosed in quotes (`"I am a string"`). If you want a string to include quotes, they must be preceded by the escape character `\` (`"I use the \"escape\" character"`). The escape character is also used to include a backslash in a string (`"The \\ character is called \"backslash\""`) and for several other purposes. Treating a string as a value is a powerful feature of C++. C++ also provides some operations on strings using which you can write some useful string expressions, but to use them you have to put

```
#include <string>
using namespace std;
```

at the beginning of your program. The `#include` directive gets you the definition of the type `string`, and the `using` declaration makes it a little more convenient to use (as will be explained later). Here are some examples:

```
string mySchool = "Bryn Mawr College";
string yourSchool = "Georgia Institute of Technology";
cout << mySchool << endl;
cout << yourSchool << endl;
cout << mySchool << " " << yourSchool << endl;
cout << yourSchool+mySchool << endl;
cout << yourSchool + ", " + mySchool << endl;
```

This is the output you would get:

```
Bryn Mawr College
Georgia Institute of Technology
Bryn Mawr College Georgia Institute of Technology
Georgia Institute of TechnologyBryn Mawr College
Georgia Institute of Technology, Bryn Mawr College
```

The operation + is defined on strings and it results in concatenating the two strings. The output command `cout <<` is followed by one or more C++ expressions, separated by <<. It evaluates all the expressions and prints out the results on the screen. As you have also seen before, this is a convenient way to print out results or messages from your program.

A Calculating Program

Ok, set your robot aside for just a few more minutes. You have now also learned enough C++ to write programs that perform simple, yet interesting, calculations. Here is a simple problem:

On January 1, 2008 the population of the world was estimated at approximately 6.650 billion people. It is predicted that at current rates of population growth, we will have over 9 billion people by the year 2050. A gross estimate of population growth puts the annual increase at +1.14% (it has been as high as +2.2% in the past). Given this data, can you estimate by how much the world's population will increase in this year (2008)? Also, by how much will it increase each day?

In order to answer the questions, all you have to do is compute 1.14% of 6.650 billion to get the increase in population this year. If you divide that number by 366 (the number of days in 2008) you will get average daily increase. You can just use a calculator to do these simple calculations, as shown below:

```
6650000000 × 1.14 / 100 = 75810000.0

75810000 / 365 = 207131.1475409836
```

That is, in this year there will be an increase of 75.81 million in the world's population which implies an average daily increase of over 207 thousand people). So now you know the answer!

You can also use C++ to do it, so let us try and write a program to do the above calculations. A program to do the calculation is obviously going to be a bit of overkill. Why do all the extra work when we already know the answer? Small steps are needed to get to higher places. So let's indulge and see how you would write a C++ program to do this. Below, we give you one version:

```
/*
 * File: worldPop.cpp
 *  Purpose:
 *      Estimate the world population growth in a year and
 *      also per day.
 *      Given that on January 1, 2008 the world's population
 *      was estimated at 6,650,000,000 and the estimated
 *      growth is at the rate of +1.14%
 */

#include <iostream>
using namespace std;

int main() {
    double population = 6650000000.0;
    double growthRate = 1.14 / 100.0;

    double growthInOneYear = population * growthRate;
    double growthInADay = growthInOneYear / 365;
```

```
    cout << "World population on January 1, 2008 is "
      << population << endl;
    cout << "By Jan. 1, 2009, it will grow by "
      << growthInOneYear << endl;
    cout << "An average daily increase of " << growthInADay
      << endl;

    return 0;
}
```

The program follows the same structure and conventions we discussed above, but there are a couple things worth mentioning. Since this program does not control the robot, we omitted `#include "Myro.h"`. However, you can see that we have added these lines:

```
#include <iostream>
using namespace std;
```

These allow us to use `cout <<` for output. We didn't need these lines in our other programs because they were included as part of `Myro.h`. We have defined variables with names `population`, and `growthRate` to designate the values given in the problem description. Although `population` is not a fractional quantity, we declared it `double`, which has to be used for large numbers (above about 2 billion). We also defined the variables `growthInOneYear` and `growthInADay` and used them to designate the results of the calculations. First, in the `main` program we define the values given, followed by

The Energy Problem

The root cause of world energy problems is growing world population and energy consumption per capita.

How many people can the earth support? Most experts estimate the limit for long-term sustainability to be between 4 and 16 billion.

From: Science, Policy & The Pursuit of Sustainability, Edited by Bent, Orr,and Baker. Illus. by Shetter. Island Press, 2002.

performing the calculation. Finally, we use the output commands to print out the result of the computations. Notice that our main program ends with:

```
return 0;
```

This is the normal way to exit a C++ main program and to signal that it completed normally. The reason you haven't seen it before is that the exit operation was performed by `disconnect()`, which we cannot use in this program.

Do This: Enter the program, and run it (just as you would run your robot programs) and observe the results. Voila! You are now well on your way to also learning the basic techniques in computing! In this simple program, we did not feel the need to define any functions. But this was a trivial program. However, it should serve to convince you that writing programs to do computation is essentially the same as controlling a robot.

Using Input

The program we wrote above uses specific values of the world's population and the rate of growth. Thus, this program solves only one specific problem for the given values. What if we wanted to calculate the results for a different growth rate? Or even a different estimate of the population? What if we wanted to try out the program for varying quantities of both? Such a program would be much more useful and could be used over and over again. Notice that the program begins by assigning specific values to the two variables:

```
double population = 6650000000.0;
double growthRate = 1.14/100.0;
```

One thing you could do is simply modify those two lines to reflect the different values. However, typical programs are much more complicated than this one and it may require a number of different values for solving a problem. When programs get larger, it is not a good idea to modify them for every specific problem instance but it is desirable to make them more useful for all problem instances. One way you can achieve this is by using the *input*

facilities of C++. All computer programs typically take some input, do some computation (or something), and then produce some output. C++ has a simple input command (defined in `<iostream>`) that can be used to rewrite the program above as follows:

```
/* File: worldPop.cpp
 *  Purpose:
 *      Estimate the world population growth in a year and
 *      also per day.
 *      Given that on January 1, 2008 the world's population
 *      was estimated at 6,650,000,000 and the estimated
 *      growth is at the rate of +1.14%
 */

#include <iostream>
using namespace std;

int main() {
    double population, growthRate, growthInOneYear,
      growthInADay;
    // print out the preamble
    cout << "This program computes population growth figures."
      << endl;
    // Input the values
    cout << "Enter current world population: ";
    cin >> population;
    cout << "Enter the growth rate: ";
    cin >> growthRate;
    growthRate = growthRate / 100.0;
    // Compute the results
    growthInOneYear = population * growthRate;
    growthInADay = growthInOneYear / 365;
    // output results
    cout << "World population today is " << population
      << endl;
    cout << "In one year, it will grow by " << growthInOneYear
      << endl;
    cout << "An average daily increase of " << growthInADay
      << endl;
    return 0;
}
```

Read the program above carefully. Look at the first line of main:

```
double population, growthRate, growthInOneYear,
  growthInADay;
```

This is an abbreviation of the following four variable declarations:

```
double population;
double growthRate;
double growthInOneYear;
double growthInADay;
```

This defines the four variables, but does not give them any initial value, since we will be reading their values from input or calculating them from input values. Notice also that we have added additional comments as well as output statements. This improves the overall readability as well as the interaction of this program. Notice also the use of the `cin >>` statements above. The basic pattern for requesting input is shown below:

```
cout << <some prompt string>;
cin >> <variable name>;
```

When the output command is executed, the computer will print out the string as a prompt. Notice that we have omitted the usual `<< endl`. This means that the output will not go to a new line, and so the input typed by the user will go on the same line as the prompt. The input command (`cin >>`) waits for the user to enter a value compatible with the variable (e.g., a number). The user can enter whatever value in response to the prompt and then hit the `RETURN` or `ENTER` key. That value is then assigned to the variable `<variable name>`. Now, look at the use of the `cin >>` command in the program above. With this modification, we now have a more general program which can be run again and again. Below, we show two sample runs:

```
*worldPop
This program computes population growth figures.
Enter current world population: 6650000000
Enter the growth rate: 1.14
World population today is 6.65e+09
In one year, it will grow by 7.581e+07
An average daily increase of 207699
*worldPop
This program computes population growth figures.
Enter current world population: 6725810000
Enter the growth rate: 2.2
World population today is 6.72581e+09
In one year, it will grow by 1.47968e+08
An average daily increase of 405391
*
```

Notice how you can re-run the program by just typing the name of the executable file. There are other ways of obtaining input in C++. We will see those a little later. Also notice the curious way some of the numbers are printed: `6.65e+09` is computerese for the scientific notation 6.65×10^9, that is, 6,650,000,000. C++ allows you to control the format in which numbers and other things are printed, but we won't worry about that just yet.

Robot Brains

Writing programs to control your robot is therefore no different from writing a program to perform a computation. They both follow the same basic structure. The only difference is that all robot programs you will write will make use of the Myro library. There will be several robot programs that will require you to obtain input from the user (see exercises below). You can then make use of the `cin >>` command as described above.

One characteristic that will distinguish robot programs from those that just do computations is in the amount of time it will take to run a program. Typically, a program that only performs some computation will terminate as soon as the computation is completed. However, it will be the case that most of the time your robot program will require the robot to perform some task over and over again. Here then, is an interesting question to ask:

Question How much time would it take for a vacuuming robot to vacuum a 16ft X 12ft room?

Seemingly trivial question but if you think about it a little more, you may reveal some deeper issues. If the room does not have any obstacles in it (i.e. an empty room), the robot may plan to vacuum the room by starting from one corner and then going along the entire length of the long wall, then turning around slightly away from the wall, and traveling to the other end. In this manner, it will ultimately reach the other side of the room in a systematic way and then it could stop when it reaches the last corner. This is similar to the way one would mow a flat oblong lawn, or even harvest a field of crop, or re-ice an ice hockey rink using a Zamboni machine. To answer the question posed above all you have to do is calculate the total distance travelled and the average speed of the vacuum robot and use the two to compute the estimated time it would take. However, what if the room has furniture and other objects in it?

You might try and modify the approach for vacuuming outlined above but then there would be no guarantee that the floor would be completely vacuumed. You might be tempted to redesign the vacuuming strategy to allow for random movements and then estimate (based on average speed of the robot) that after some generous amount of time, you can be assured that the room would be completely cleaned. It is well known (and we will see this more formally in a later chapter) that random movements over a long period of time do end up providing uniform and almost complete coverage. Inherently this also implies that the same spot may very well end up being vacuumed several times (which is not necessarily a bad thing!). This is similar to the thinking that a herd of sheep, if left grazing on a hill, will result, after a period of time, in a nearly uniform grass height (think of the beautiful hills in Wales).

On the more practical side, iRobot's Roomba robot uses a more advanced strategy (though it is time based) to ensure that it provides complete coverage. A more interesting (and important) question one could ask would be:

Question: How does a vacuuming robot know that it is done cleaning the room?

Most robots are programmed to either detect certain terminating situations or are run based on time. For example, run around for 60 minutes and then stop. Detecting situations is a little difficult and we will return to that in the next chapter.

So far, you have programmed very simple robot behaviors. Each behavior which is defined by a function, when invoked, makes the robot do something for a fixed amount of time. For example, the `yoyo` behavior from the last chapter when invoked as:

```
yoyo(0.5, 1);
```

would cause the robot to do something for about 2 seconds (1 second to go forward and then 1 second to move backward). In general, the time spent carrying out the `yoyo` behavior will depend upon the value of the second parameter supplied to the function. Thus if the invocation was:

```
yoyo(0.5, 5.5);
```

the robot would move for a total of 11 seconds. Similarly, the `dance` behavior defined in the previous chapter will last a total of six seconds. Thus, the total behavior of a robot is directly dependent upon the time it would take to execute all the commands that make up the behavior. Knowing how long a behavior will take can help in pre-programming the total amount of time the overall behavior could last. For example, if you wanted the robot to perform the dance moves for 60 seconds, you can repeat the `dance` behavior ten times. You can do this by simply issuing the `dance` command 10 times. But that gets tedious for us to have to repeat the same commands so many times. Computers are designed to do repetitious tasks. In fact, repetition is one of the key concepts in computing and all programming languages, including C++, provide simple ways to specify repetitions of all kinds.

Doing Repetition in C++

If you wanted to repeat the dance behavior 10 times, all you have to do is:

```
for (int i = 1; i <= 10; i++) {
   dance();
}
```

(Notice that, as is often the case, the right curly brace is *not* followed by a semicolon.) This is a new statement in C++: the `for`-statement. It is also called a *loop statement* or simply a *loop*. It is a way of repeating something a fixed number of times. The basic syntax of a `for`-loop in C++ is:

```
for (<initialization>; <continuation>; <incrementation>) {
   <do something>
   <do something>
   ...
}
```

The loop specification begins with the keyword `for` which is followed by three expressions separated by semicolons and all surrounded by parentheses. This line sets up the number of times the repetition will be repeated. What follows is a set of statements surrounded by braces, which are called a *block* that forms the *body* of the loop (stuff that is repeated).

The first expression is the `<initialization>`, which typically defines an integer variable and gives it an initial value, for example, `int i = 1`. The `<continuation>` specifies a condition, and the loop will continue to repeat so long as the condition is true. For example, in the previous example the continuation condition is `i <= 10`, and so the loop will continue to repeat so long as the value of variable `i` is less than or equal to 10. ("`<=`" is a common way of writing "less than or equal to" in programming languages.) The third expression is the `<incrementation>`, which is a command that is executed at the end of each pass through the loop. In this case it is `i++`, which is a C++ abbreviation for "increment `i`" ("increase the value of `i` by one"). (Since C++ is a successor to a programming language named "C," you can probably guess

how it got its name!) So our example `for` loop works like this. First it does `int i = 1`, which declares an integer variable called `i` and gives it the initial value 1. Since the continuation condition `i <= 10` is satisfied, it executes the commands in the loop body, which in this case is just `dance()`. When it gets to the end of the loop body, it executes `i++`, which increases the value of `i` to 2. It checks the continuation condition, and since `i` is still less than 10, it executes the loop body a second time. This continues, with the loop body being executed 10 times. However, when it gets to the end of the tenth pass, the `i++` command increases `i` to 11, and so the continuation condition is not satisfied and the repetition stops. Thus `i` (which is called a *loop index variable*) is assigned successive values in the sequence 1, 2, …, 10, and for each of those values, the statements in the body of the loop are executed.

Do This: Let us try this out on the robot. Modify the robot program from the start of this chapter to include the dance function and then write a main program to use the loop above.

```
/*
 * File: dance.cpp
 * Purpose: A simple dance routine
 */

// First include standard declarations
#include "Myro.h"

// Define the new functions...

void yoyo(double speed, double waitTime) {
       robot.forward(speed, waitTime);
       robot.backward(speed, waitTime);
}

void wiggle(double speed, double waitTime) {
       robot.motors(-speed, speed);
       wait(waitTime);
       robot.motors(speed, -speed);
       wait(waitTime);
       robot.stop();
}
```

```
void dance() {
    yoyo(0.5, 0.5);
    yoyo(0.5, 0.5);
    wiggle(0.5, 1);
    wiggle(0.5, 1);
}

// The main dance program
int main() {
       connect();
       cout << "Running the dance routine..." << endl;
       for (int danceStep = 1, danceStep <= 10, danceStep++) {
           dance();
       }
       cout << "...Done" << endl;
       disconnect();
}
```

Notice that we have used `danceStep` (a more meaningful name than `i`) to represent the loop index variable. When you run this program, the robot should perform the dance routine ten times. Modify the value specified in the `for` command to try out some more steps. If you end up specifying a really large value, remember that for each value of `danceStep` the robot will do something for 6 seconds. Thus, if you specified 100 repetitions, the robot will run for 10 minutes.

In addition to repeating by counting, you can also specify repetition using time. For example, if you wanted the robot (or the computer) to do something for 30 seconds. You can write the following command to specify a repetition based on time:

```
while (timeRemaining(10)) {
    <do something>
    <do something>
    ...
}
```

The above commands will be repeated for 10 seconds. Thus, if you wanted the computer to say "Doh!" for 5 seconds, you can write:

```
while (timeRemaining(5)) {
    speak("Doh!", 0);
}
```

In writing robot programs there will also be times when you just want the robot to keep doing its behaviors forever! While technically by *forever* we do mean eternity in reality what is likely to happen is either it runs out of batteries, or you decide to stop it (by aborting the program). The C++ command to specify this uses a different loop statement, called a `while`-loop that can be written as:

```
while (true) {
   <do something>
   <do something>
   ...
}
```

`true` is also a value in C++ (along with `false`) about which we will learn more a little later. For now, it would suffice for us to say that the above loop is specifying that the body of the loop be executed forever!

Do This: Modify the `dance.cpp` program to use each of the `while`-loops instead of the `for`-loop. In the last case (`while (true)`) remember to abort the program to stop the repetitions (and the robot).

As we mentioned above, repetition is one of the key concepts in computing. For example, we can use repetition to predict the world population in ten years by repeatedly computing the values for each year:

```
for (int year = 1; year <= 10; year++) {
    population = population * (1 + growthRate)]
}
```

That is, repeatedly add the increase in population, based on growth rate, ten times.

Do This: Modify the `worldPop.cpp` program to input the current population, growth rate, and the number of years to project ahead and compute the resulting total population. Run your program on several different values (Google: "world population growth" to get latest numbers). Can you estimate when the world population will become 9 billion?

Summary

This chapter introduced the basic structure of C++ (and robot) programs. We also learned about *names* and *values* in C++. Names can be used to designate functions and values. The latter are also called *variables*. C++ provides several different types of values: integers, floating point numbers, strings, and also boolean values (`true` and `false`). Most values have built-in operations (like addition, subtraction, etc.) that perform calculations on them. C++ provides simple built-in facilities for obtaining input from the user. All of these enable us to write not only robot programs but also programs that perform any kind of computation. Repetition is a central and perhaps the most useful concept in computing. In C++ you can specify repetition using either a `for`-loop or a `while`-loop. The latter are useful in writing general robot brain programs. In later chapters, we will learn how to write more sophisticated robot behaviors.

Myro Review

`speak(<something>);`
The computer converts the text in `<something>` to speech and speaks it out. `<something>` is also simultaneously printed on the screen. Speech generation is done synchronously. That is, anything following the speak command is done only after the entire thing is spoken.

`speak(<something>, 0);`
The computer converts the text in `<something>` to speech and speaks it out.

`<something>` is also simultaneously printed on the screen. Speech generation is done asynchronously. That is, execution of subsequent commands can be done prior to the text being spoken.

`timeRemaining(<seconds>)`
This is used to specify timed repetitions in a while-loop (see below).

C++ Review

Values

Values in C++ include numbers (integers or floating point numbers) and strings. Each type of value can be used in an expression by itself or using a combination of operations defined for that type (for example, `+`, `-`, `*`, `/`, `%` for numbers). Strings are considered sequences of characters (or letters).

Names

A name in C++ must begin with either an alphabetic letter (`a-z` or `A-Z`) or the underscore (i.e. _) and can be followed by any sequence of letters, digits, or underscore letters.

```
cin >> <variable>;
```

Waits for the user to type a value compatible with the type of the variable. This value becomes the new value of the variable.

```
#include "Myro.h"
<any other includes>
<function definitions>
int main() {
  connect();
  <do something>
  <do something>
  ...

  disconnect();
} // end of main program
```

This is the basic structure of a robot control program in C++. Without the `#include` line and the `connect` and `disconnect` commands, it is the basic structure of all C++ programs.

`cout << <expression> << <expression> << … << <expression>;`
This function prints out the values of the expressions (one or more) in the console window. If any of the expressions are `endl`, the output will go to a new line at that point.

`<type> <variable name> = <expression>;`
This is how C++ defines a variable of the specified type (e.g., `double`, `int`, `string`). The value generated by `<expression>` will become the initial value of `<variable name>`.

`<variable name> = <expression>;`
This is how C++ assigns values to variables. The value generated by <expression> will become the new value of <variable name>.

`<expression> <= <expression>`
This condition is true if the first expression is less than or equal to the second. Other relations include >= (greater or equal), <, >, == (equal) and != (not equal).

`<variable>++`
Increments <variable>, that is, increases the value of <variable> by 1.

Repetition

```
for (<initialization>; <continuation>; <incrementation>) {
   <do something>
   <do something>
   ...
}
```

```
while (timeRemaining(<seconds>)) {
    <do something>
    <do something>
    ...
}

while (true) {
   <do something>
   <do something>
   ...
}
```

These are different ways of doing repetition in C++. The first version will initialize a variable and repeat the loop body, executing the `<incrementation>` at the end of each repetition, and continue repeating so long as the continuation condition remains true. The second version will carry out the body for `<seconds>` amount of time. `timeRemaining` is a Myro function (see above). The last version specifies an un-ending repetition.

Exercises

1. Write a C++ program to convert a temperature from degrees Celsius to degrees Fahrenheit. Here is a sample interaction with such a program:

```
Enter a temperature in degrees Celsius: 5.0
That is equivalent to 41.0 degrees Fahrenheit.
```

The formula to convert a temperature from Celsius to Fahrenheit is: `C/5=(F-32)/9`, where `C` is the temperature in degrees Celsius and `F` is the temperature in degrees Fahrenheit.

2. Write a C++ program to convert a temperature from degrees Fahrenheit to degrees Celsius.

3. Write a program to convert a given amount of money in US dollars to an equivalent amount in Euros. Look up the current exchange rate on the web (see xe.com, for example).

4. Modify the version of the dance program above that uses a `for`-loop to use the following loop:

```
for (int danceStep = 0; danceStep < 10; danceStep++) {
    dance();
}
```

5. Run the world population program (any version from the chapter) and when it prompts for input, try entering the following and observe the behavior of the program. Also, given what you have learned in this chapter, try and explain the resulting behavior.

a. Use the values 9000000000, and 1.42 as input values as above. Except, when it asks for various values, enter them in any order. What happens?

b. For any of the values to be input, replace them with a string. For instance enter `"Al Gore"` when it prompts you for a number. What happens?

6. Rewrite your solution to Exercise 4 from the previous chapter to use the program structure described above.

7. You were introduced to the rules of naming in C++. You may have noticed that we have made extensive use of *mixed case* in naming some entities. For example, `waitTime`. There are several naming conventions used by programmers and that has led to an interesting culture in of itself. Look up the phrase *CamelCase controversy* in your favorite search engine to learn about naming conventions. For an interesting article on this, see The Semicolon Wars (www.americanscientist.org/issues/pub/the-semicolon-wars).

8. Experiment with the `speak` function introduced in this chapter. Try giving it a number to speak (try both integers and floating point numbers). What is the largest integer value that it can speak? What happens when this limit is exceeded? Try to give the `speak` function a list of numbers, or strings, or both.

9. Write a C++ program that sings the ABC song: *ABCD…XYZ. Now I know my ABC's. Next time won't you sing with me?*