

Behavior Control

Oh, Behave!

Austin Powers (played by Mike Myers) in the movie
Austin Powers: International Man of Mystery,
New Line Cinema, 1997.

Opposite page: My Traffic Lights
Illustration by Yingshun Wong (yingshun.co.uk)

Writing programs is all about exercising control. In the case of a robot's brain your program directs the operations of a robot. However, it is also important to realize that the program itself is really controlling the computer. That is, when you write C++ programs you are controlling the computer that is then communicating with the robot. Your program is directing the computer to control the robot. If you take Myro out of the picture you are writing programs to control the computer. This is the sense in which learning with robots also leads to learning computing. Every program you write is doing computation. Contrary to popular misconceptions computing is not just about doing calculations with numbers. Controlling your robot is also computing, as is predicting the world's population, or composing an image, etc. This is one aspect of control.

When writing robot control programs, the structure you use to organize the program itself is a control strategy. Programming a robot is specifying automated control. As a programmer or behavior designer you structure your program to accomplish the goals of the behavior: how the sensors are used to decide what to do next. This is another aspect of control. So far, you have seen how to write control programs using Braitenberg style sensor-motor *wiring*. You have also seen how to specify reactive control. These are examples of two robot control paradigms.

In this chapter we delve further into the world of computation and robot control paradigms. We will learn how to write robot control programs for more complex and more robust robot tasks. We will also see how, using the concepts learned so far, we can write useful and interesting computer applications.

Behavior-based Control

When writing robot control programs, so far, you have used a very basic technique in designing control programs:

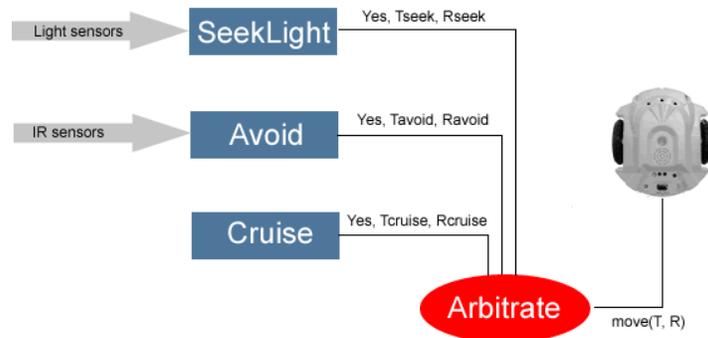
```
int main() {  
    // do forever or for some time  
    // or until a certain condition is satisfied  
  
    // sense and transform sensor values  
    // reason or decide what to do next  
    // do it  
}
```

As you have seen, such a control program works well for many simple tasks. However, you may have already run into many situations where, once the task gets a little complex, it becomes difficult to structure a program in terms of a single stream of control as shown above. For example, the corral exiting behavior from the last chapter requires you to combine two simple behaviors: solve a maze (avoid obstacles) and seek light to get out of the corral. As you have seen before, it is fairly easy to program each of the individual behaviors: obstacle avoidance; light following. But, when you combine these behaviors to accomplish the corral exiting behavior two things happen: you are forced to amalgamate the two control strategies into a single one and it may become difficult to decide which way to combine them; additionally, the resulting program is not very pretty and hard to read. In reality, hardly any robot programs are written that way. In this section, we will look at a different way of structuring robot programs that makes designing behaviors easy, and yet, the resulting structure of the overall program is also clean and straightforward. You can design some very sophisticated behaviors using these ideas.

People in the robotics community call the style of programming shown above as *reactive control* or *direct control*. Also referred to as *sensor fusion*, the resulting programs are purely sensor driven and hence appear to be too bottom-up. That is, the values of the sensors drive the logic of control as opposed to the goals of the robot tasks themselves. In *behavior-based control* you get away from sensors and focus the design of your robot programs based on the number and kinds of behaviors your robot has to carry out.

Let us look at how behavior-based control enables us to design the corral exiting behavior. The robot essentially has to carry out three kinds of behaviors: cruise (in the absence of any obstacles and/or light), avoid

obstacles (if present), and seek light (if present). In a behavior-based style of writing programs, you will define each of these behaviors as an individual decision unit. Thus, each is quite simple and straightforward to write. Next, the control program has to fuse the behaviors recommended by each individual behavior unit. Look at the picture shown below:



In the diagram above, we have shown the three basic behaviors that we are trying to combine: *Cruise*, *Avoid*, *SeekLight*. Each of these behaviors outputs a triple: *Yes/No*, *Translate Speed*, *Rotate Speed*. A *Yes* implies that the behavior module has a recommendation. *No* implies that it doesn't. That is, it allows the possibility of a behavior having no recommendation. For example, in the corral exiting situation, in the absence of a light source being sensed by the robot, the *SeekLight* module will not have any recommendation. It then becomes the task of the arbitrator (or the decision module) to decide which of the available recommendations to use to drive the robot. Notice that in the end to control the robot, all one has to do is decide how much to translate and rotate. Many different arbitration schemes can be incorporated. We will use a simple but effective one: Assign a priority to each behavior module. Then the arbitrator always chooses the highest priority recommendation. This style of control architecture is also called *subsumption architecture*. In the figure above, we have actually drawn the modules in the order of their priority: the higher the module is in the figure, the higher its priority. The lowest behavior,

cruise, does not require any sensors and is always present: it wants the robot to always go forward.

Arranging a control based on combining simple behaviors has several advantages: you can design each individual behavior very easily; you can also test each individual behavior by only adding that behavior and seeing how well it performs; you can incrementally add any number of behaviors on top of each other. In the scheme above, the control regime implies that the robot will always cruise forward. But, if there is an obstacle present, it will override the cruise behavior and hence try to avoid the obstacle. However, if there is a light source detected, it will supersede all behaviors and engage in light seeking behavior. Ideally, you can imagine that all the behaviors will be running simultaneously (asynchronously). In that situation, the arbitrator will always have one or more recommendations to adopt based on priority.

Let us develop the program that implements behavior-based control. First, we define each behavior to possibly recommend an action. To do this, we use a C++ feature that you have not seen before; it is called a *structure*:

```
struct rec {
    bool active;
    double translateSpeed;
    double rotateSpeed;
};
```

In effect `struct` defines a new data type, which is somewhat like a vector, in that it groups several data values together, but unlike a vector in that the values do not have to be of the same type. In this case, the `rec` structure is composed of three *fields*, a Boolean called `active`, and two doubles called `translateSpeed` and `rotateSpeed`. The intent is that if `active` is `true`, then the structure represents an active recommendation that the robot move forward at rate `translateSpeed` while turning at rate `rotateSpeed`.

To make a recommendation, we create a structure of type `rec` and initialize its fields, which can be accomplished with the function:

```
rec make_rec (bool a, double t, double r) {
    rec R;
    R.active = a;
    R.translateSpeed = t;
    R.rotateSpeed = r;
    return R;
}
```

This function has three arguments (a Boolean and two doubles) and it returns a structure of type `rec`. To do this it declares a variable `R` of type `rec`, and initializes its three fields to the supplied parameters. Notice that the fields of `R` are named `R.active`, `R.translateSpeed`, and `R.rotateSpeed`. Given these definitions, each of the desired behaviors can recommend an action:

```
double cruiseSpeed = 0.8;
double turnSpeed = 0.8;
int lightThresh = 80;

rec cruise () {
    // is always active, just move forward
    return make_rec (true, cruiseSpeed, 0);
}

rec avoid () {
    // see if there are any obstacles
    int L = 1 - robot.getIR("left");
    int R = 1 - robot.getIR("right");
    if (L) {
        return make_rec (true, 0, -turnSpeed);
    } else if (R) {
        return make_rec (true, 0, turnSpeed);
    } else {
        return make_rec (false, 0, 0);
    }
}

rec seekLight () {
    int L = robot.getLight("left");
    int R = robot.getLight("right");
}
```

```
if (L < lightThresh) {
    return make_rec (true, cruiseSpeed/2.0, turnSpeed);
} else if (R < lightThresh) {
    return make_rec (true, cruiseSpeed/2.0, -turnSpeed);
} else {
    return make_rec (false, 0, 0);
}
}
```

In the above, you can see that each individual behavior is simple and is easy to read (and write). Also, notice that the behaviors do not always recommend an action. If the first parameter to `make_rec` is `false`, then the behavior is not recommending any action (it is inactive); in this case, the behavior has not been *released* by the stimuli (sensor values). There are several ways to incorporate these behaviors into a behavior-based program. Here is one:

```
int main () {
    connect ();

    while (true) {
        rec decision = arbitrate ();
        robot.move (decision.translateSpeed,
                    decision.rotateSpeed);
    }

    disconnect ();
}
```

The main program calls the `arbitrate` function to decide which recommendation to take, which is saved in the variable `decision`. Then the motor behavior (translation and rotation speeds) from the chosen recommendation is applied to the robot. The function `arbitrate` is simple enough; it returns the first (highest priority) active recommendation:

```
rec arbitrate () {  
    rec R;  
    R = seekLight(); if (R.active) return R;  
    R = avoid();     if (R.active) return R;  
    R = cruise();    if (R.active) return R;  
}
```

For readability, we have written several things a little differently than previously in this book. First, we have written three statements (an assignment, an `if`, and a `return`) on a single line, to emphasize the close relationship among them. As previously mentioned, C++ really doesn't care how many or how few lines are used to write something. Second, the `return`-statements, which are the bodies of the `if`-statements, are not surrounded by curly braces. As it turns out, all the C++ statements that govern a body (e.g., `for`, `else`, `while`, function definitions) allow the curly braces to be omitted if the body is a single statement; in effect the braces act like parentheses to group a set of statements into a body. However, it is easier to maintain programs if you always use curly braces, even when they are not strictly necessary, since otherwise you may forget to put them in if you change a single-statement body to a multiple-statement body. As in prose, style is important in programming, and coding conventions such as this one make programs easier to read and to maintain.

Do This: Implement the program above and see how well the robot behaves in navigating around and exiting the corral. What happens if you change the priority (ordering in the list) of behaviors?

The C++ Math Library

Most programming languages, C++ included, provide a healthy selection of libraries of useful functions so you do not have to write them. Such libraries are often termed as an *application programming interface* or an *API*. Earlier you were introduced to the `cstdlib` library provided by C++. It contains several useful functions that provide facilities for generating random numbers. Similarly, C++ also provides a `cmath` library that provides frequently used mathematical functions. In Chapter 6, we used the function `exp` from the

`cmath` library to normalize sensor values. Some of the commonly used functions in the `cmath` library are listed below:

Commonly used functions in the `cmath` library

`ceil(x)` Returns the ceiling of x as a float, the smallest integer value greater than or equal to x .

`floor(x)` Returns the floor of x as a float, the largest integer value less than or equal to x .

`exp(x)` Returns e^x .

`log(x[, base])` Returns the logarithm of x to the given base. If the base is not specified, return the natural logarithm of x (i.e., $\log_e x$).

`log10(x)` Returns the base-10 logarithm of x (i.e. $\log_{10} x$).

`pow(x, y)` Returns x^y .

`sqrt(x)` Returns the square root of x (\sqrt{x}).

Some of the other functions available in the `cmath` library are listed at the end of the chapter. In order to use any of these all you have to do is include the `cmath` library:

```
#include <cmath>
```

```
cout << ceil(5.34) << endl;
```

```
6
```

```
cout << floor(5.34) << endl;
```

```
5
```

```
cout << exp(3) << endl;
```

```
20.0855
```

```
cout << log10(1000) << endl;
```

```
3
```

```
cout << log(1024, 2) << endl;
```

```
10
```

```
cout << pow(2, 10) << endl;
```

```
1024
```

```
cout << sqrt(7.0) << endl;
```

```
2.64575
```

Doing Computations

Lets us weave our way back to traditional style computing for now. You will see that the concepts you have learned so far will enable you to write lots of different and more interesting computer applications. It will also give you a clear sense of the structure of typical computer programs. Later, in Chapter 11, we will also return to the larger issue of the design of general computer programs.

A Loan Calculator

Your current car, an adorable 1992 SAAB 93 was bought used and, for past several months, you have had nothing but trouble keeping the car on the road. Last night the ignition key broke off in the key slot when you were trying to start it and now the broken piece would not come out (this used to happen a lot with older SAAB's). The mechanic has to dismantle the entire ignition assembly to get the broken key out and it could cost you upwards of \$500. The car's engine, which has done over 185,000 miles, has left you stranded on the road many times. You have decided that this is it; you are going to go out and get yourself a brand new reliable car. You have been moonlighting at a restaurant to make extra money and have managed to save \$5500.00 for exactly this situation. You are now wondering what kind of new car you can buy. Obviously, you will have to take a loan from a bank to finance the rest of the cost but you are not sure how big a loan, and therefore what kind of car,

you can afford. You can write a small C++ program to help you try out various scenarios.

You can either dream (realistically) of the car you would like to buy, or you can go to any of the helpful car buying sites on the web (www.edmunds.com is a good place to start). Let us say that you have spent hours looking at features and options and have finally narrowed your desires to a couple of choices. Your first choice is going to cost you \$22,000.00 and your second choice is priced at \$18,995.00. Now you have to decide which of these you can actually afford to purchase.

First, you go talk to a couple of banks and also look at some loan offers on the web. For example, go to bankrate.com and look for current rates for new car loans.

Suppose the loan rates quoted to you are: 6.9% for 36 months, 7.25% for 48 months, and 7.15 for 60 months.

You can see that there is a fair bit of variation in the rates. Given all this information, you are now ready to write a program that can help you figure out which of the two choices you may be able to make. In order to secure the loan, you have to ensure that you have enough money to pay the local sales tax (a 6% sales tax on a \$20,000 car will add up to a hefty \$1200!). After paying the sales tax you can use the remainder of the money you have saved up towards the down payment. The remainder of the money is the amount that you would borrow. Depending on the type of loan you choose, your monthly payments and how long you will make those payments will vary. There is a simple formula that you can use to estimate your monthly payment:

$$\text{MonthlyPayment} = \frac{\text{LoanAmount} * \text{MonthlyInterestRate}}{1 - e^{-\text{LoanTerm} * \log(1 + \text{MonthlyInterestRate})}}$$

Whoa! That seems complicated. However, given the formula, you can see that it really requires two mathematical functions: $\log(x)$ and e^x , both of which are

available in the C++ `cmath` library. Suddenly, the problem seems not that hard.

Let us try and outline the steps needed to write the program: First, note the cost of the car, the amount of money you have saved, and the sales tax rate. Also, note the financials: the interest rate, and the term of the loan. The interest rate quoted is generally the annual percentage rate (APR) convert it to monthly rate (by dividing it by 12). Next, compute the sales tax you will pay. Use the money left to make a down payment. Then determine the amount you will borrow. Plug in all of the values in the formula and compute the monthly payment. Also, compute the total cost of the car. Output all the results. Next, we can take each of the above steps and start to encode them into a program. Here is a first order refinement:

```
int main() {
    // First, note the cost of the car (Cost),
    // the amount of money you have saved (Cash),
    // and the sales tax rate (TaxRate) (6% e.g.)

    // Also, note the financials: the interest rate (APR),
    // and the term of the loan (Term)
    // The interest rate quoted is generally the annual
    // percentage rate (APR)
    // Convert it (APR) to monthly rate (divide it by 12) (MR)

    // Next, compute the sales tax you will pay (SalesTax)
    // Use the money left to make a down payment (DownPayment)
    // Then determine the amount you will borrow (LoanAmount)

    // Plug in all of the values in the formula and compute
    // the monthly payment (MP)

    // Also, compute the total cost of the car. (TotalCost)

    // Output all the results
}
```

Above, we have taken the steps and converted them into a skeletal C++ program. All the steps are converted to C++ comments and where needed, we

have decided the names of variables that will hold the values that will be needed for the calculations. This is useful because this also helps determine how the formula will be encoded and also helps determine what values can be programmed in and which ones you will have to supply as input. Making the program require inputs will easily enable you to enter the different parameters and then based on the outputs you get, you can decide which car to buy. Let us encode all the inputs first:

```
int main() {
    // Input variables
    double Cost, Cash, APR;
    int term;

    // First, note the cost of the car (Cost),
    cout << "Enter the cost of the car: $";
    cin >> Cost;

    // the amount of money you have saved (Cash),
    cout << "Enter the amount of money you saved: $";
    cin >> Cash;

    // and the sales tax rate (TaxRate) (6% e.g.)
    double SalesTaxRate = 6.0;

    // Also, note the financials: the interest rate (APR),
    // and the term of the loan (Term)
    // The interest rate quoted is generally the annual
    // percentage rate (APR)
    cout << "Enter the APR for the loan (in %): ";
    cin >> APR;

    // Convert it (APR) to monthly rate (divide it by 12) (MR)

    // Next, compute the sales tax you will pay (SalesTax)
    // Use the money left to make a down payment (DownPayment)
    // Then determine the amount you will borrow (LoanAmount)

    // Plug in all of the values in the formula and compute
    // the monthly payment (MP)

    // Also, compute the total cost of the car. (TotalCost)
```

```
    // Output all the results  
}
```

We have refined the program to include the inputs that will be needed for each run of the program. Notice that we chose not to input the sales tax rate and instead just assigned it to the variable `SalesTaxRate`. If you wanted, you could also have that be entered as input. What you choose to have as input to your program is your design decision. Sometimes the problem may be framed so it explicitly specifies the inputs; sometimes you have to figure that out. In general, whatever you need to make your program more versatile is what you have to base your decisions on. For instance, fixing the sales tax rate to 6.0 will make the program usable only in places where that rate applies. If, for example, you wanted your friend in another part of the country to use the program, you should choose to make that also an input value. Let us go on to the next steps in the program and encode them in C++. These are mainly computations. The first few are simple. Perhaps the most complicated computation to encode is the formula for computing the monthly payment. All of these are shown in the version below.

```
#include <iostream>  
#include <cmath>  
using namespace std;  
  
int main() {  
    // Input variables  
    double Cost, Cash, APR;  
    int term;  
  
    // First, note the cost of the car (Cost),  
    cout << "Enter the cost of the car: $";  
    cin >> Cost;  
  
    // the amount of money you have saved (Cash),  
    cout << "Enter the amount of money you saved: $";  
    cin >> Cash;  
  
    // and the sales tax rate (TaxRate) (6% e.g.)  
    double SalesTaxRate = 6.0;  
  
    // Also, note the financials: the interest rate (APR),
```

```
// and the term of the loan (Term)
// The interest rate quoted is generally the annual
// percentage rate (APR)
cout << "Enter the APR for the loan (in %): ";
cin >> APR;

// Input the term of the loan (Term)
cout << "Enter length of loan term (in months): ";
cin >> term;

// Convert it (APR) to monthly rate (divide it by 12) (MR)
// also divide it by 100 since the value input is in %
double MR = APR/12.0/100.0;

// Next, compute the sales tax you will pay (SalesTax)
double SalesTax = Cost * SalesTaxRate / 100.0;

// Use the money left to make a down payment (DownPayment)
double DownPayment = Cash - SalesTax;

// Then determine the amount you will borrow (LoanAmount)
double LoanAmount = Cost - DownPayment;

// Plug in all of the values in the formula and compute
// the monthly payment (MP)
double MP = (LoanAmount * MR)
            / (1.0 - exp(-term * log(1.0+MR)));

// Also, compute the total cost of the car. (TotalCost)
double TotalCost = SalesTax + DownPayment + MP * term;

// Output all the results
cout << "Here are the details about your new car...\n";
cout << "-----\n";
cout << endl;
cout << "Money you have saved $" << Cash << endl;
cout << "Cost of the car $" << Cost << endl;
cout << "Sales Tax rate is " << SalesTaxRate << "%\n";
cout << "Sales Tax on the car $" << SalesTax << endl;
cout << "Your down payment will be $" << DownPayment
    << endl;
cout << "You will be borrowing $" << LoanAmount << endl;
cout << "A " << term << " month loan at " << APR
```

```
    << "% APR\n";
    cout << "Your monthly payment will be $" << MP << endl;
    cout << "Total cost will be $" << TotalCost << endl;
    cout << endl;
    return 0;
}
```

Notice that some of the output strings contain the special “escape sequence” “\n” (“newline”), which means “go to a new line,” and thus has the same effect as `<< endl;` sometimes it’s more convenient, as shown here.

Do This: When you enter the above C++ program and run it, you can enter the data about your car. Here is a sample run:

```
Enter the cost of the car: $20000.00
Enter the amount of money you saved: $5500.00
Enter the APR for the loan (in %): 6.9
Enter the length of loan term (in months): 36
Here are the details about your new car...
-----

Money you have saved $5500
Cost of the car $20000
Sales Tax rate is 6%
Sales Tax on the car $1200
Your down payment will be $4300
You will be borrowing $15700
A 36 month loan at 6.9% APR
Your monthly payment will be $484.053
Total cost will be $22925.9
```

It appears that at for the \$20000.00 car, for a 36 month 6.9% loan you will end up paying \$484.05 (or \$484.06 depending upon how your loan company rounds pennies!).

When you need to restrict your output values to specific decimal places (two in the case of dollars and cents, for example), you can use the `printf` formatted output function built into C++. For example, using a format string, you can specify how to output a fixed-point value as follows:

```
double pi = acos(-1); // this computes the value of pi

printf ("Value of PI %5.3f in 3 places after the decimal\n",
       pi);
```

The above is a C++ expression that has the syntax:

```
printf (<string>, <expression>, <expression>, ...);
```

The `<expression>`s are printed according to the format `<string>`. Inside `<string>` there is a format specification beginning with a `%`-sign and ending in an `f`. What follows the `%`-sign is a numerical specification of the value to be inserted at that point in the string. The `f` in the specification refers to the fact it is for a floating point value. Between the `%`-sign and the `f` is a number, `5.3`. This specifies that the floating point number to be inserted will take up at least 5 spaces, 3 of which will be after the decimal. One of the spaces is always occupied by the decimal. Thus, the specification `%5.3f` specifies a value to be printed in the following format:

```
Value of PI -.--- expressed in 3 places after the decimal
```

You can see the result of executing this in C++ below:

```
printf ("Value of PI %5.3f in 3 places after the decimal\n",
       pi);
```

```
Value of PI 3.142 in 3 places after the decimal
```

```
printf ("Value of PI %7.4f in 4 places after the decimal\n",
       pi);
```

```
Value of PI 3.1416 in 4 places after the decimal
```

In the second example above, we replaced the specification with `%7.4f`. Notice that the resulting string allocates seven spaces to print that value. If there are more spaces than needed they get padded by blanks on the leading edge (notice the extra space before 3 in the second example above). If the

space specified is less, it will always be expanded to accommodate the number. For example:

```
printf ("Value of PI %1.4f in 4 places after the decimal\n",  
pi);
```

```
Value of PI 3.1416 in 4 places after the decimal
```

We deliberately specified that the value be 1 space wide with 4 spaces after the decimal (i.e. `%1.4f`). As you can see, the space was expanded to accommodate the value. What is assured is that the value is always printed using the exact number of spaces after the decimal. Here is another example:

```
printf ("5 is also %1.3f with 3 places after the decimal.\n",  
5.0);
```

```
5 is also 5.000 with 3 places after the decimal.
```

Thus, the value is printed as `5.000` (i.e. the three places after the decimal are always considered relevant in a specification like `%1.3f`). Similarly, for specifying whole number or integer values you can use the letter-`d`, and for strings you can use the letter-`s`:

```
printf ("Hello %10s, how are you?\n", "Arnold");
```

```
Hello      Arnold, how are you?
```

By default, longer specifications are right-justified. You can use a `%-` specification to left-justify. For example:

```
printf ("Hello %-10s, how are you?\n", "Arnold");
```

```
Hello Arnold , how are you?
```

Having such control over printed values is important when you are trying to output tables of aligned values. Let us modify our program from above to use these formatting features:

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

int main() {
    // Input variables
    double Cost, Cash, APR;
    int term;

    // First, note the cost of the car (Cost),
    cout << "Enter the cost of the car: $";
    cin >> Cost;

    // the amount of money you have saved (Cash),
    cout << "Enter the amount of money you saved: $";
    cin >> Cash;

    // and the sales tax rate (TaxRate) (6% e.g.)
    double SalesTaxRate = 6.0;

    // Also, note the financials: the interest rate (APR),
    // and the term of the loan (Term)
    // The interest rate quoted is generally the annual
    // percentage rate (APR)
    cout << "Enter the APR for the loan (in %): ";
    cin >> APR;

    // Input the term of the loan (Term)
    cout << "Enter length of loan term (in months): ";
    cin >> term;

    // Convert it (APR) to monthly rate (divide it by 12) (MR)
    // also divide it by 100 since the value input is in %
    double MR = APR/12.0/100.0;

    // Next, compute the sales tax you will pay (SalesTax)
    double SalesTax = Cost * SalesTaxRate / 100.0;

    // Use the money left to make a down payment (DownPayment)
    double DownPayment = Cash - SalesTax;

    // Then determine the amount you will borrow (LoanAmount)
    double LoanAmount = Cost - DownPayment;
```

```
// Plug in all of the values in the formula and compute
// the monthly payment (MP)
double MP = (LoanAmount * MR)
           / (1.0 - exp(-term * log(1.0+MR)));

// Also, compute the total cost of the car. (TotalCost)
double TotalCost = SalesTax + DownPayment + MP * term;

// Output all the results
printf ("Here are the details about your new car...\n");
printf ("-----\n");
printf ("\n");
printf ("Money you have saved $%1.2f\n", Cash);
printf ("Cost of the car $%1.2f\n", Cost);
printf ("Sales Tax rate is %1.2f%%\n", SalesTaxRate);
printf ("Sales Tax on the car $%1.2f\n", SalesTax);
printf ("Your down payment will be $%1.2f\n",
        DownPayment);
printf ("You will be borrowing $%1.2f\n", LoanAmount);
printf ("A %2d month loan at %1.2f%% APR\n", term, APR);
printf ("Your monthly payment will be $%1.2f\n", MP);
printf ("Total cost will be $%1.2f\n", TotalCost);
printf ("\n");
return 0;
}
```

Notice that we used %% in the format string to print a percent sign after the APR. When you run the program again (say for a slightly different loan term), you get:

```
Enter the cost of the car: $20000.00
Enter the amount of money you saved: $5500.00
Enter the APR for the loan (in %): 7.25
Enter the length of loan term (in months): 48
Here are the details about your new car...
-----
```

```
Money you have saved $5500.00
Cost of the car $20000.00
Sales Tax rate is 6.00%
Sales Tax on the car $ 1200.00
Your down payment will be $4300.00
You will be borrowing $15700.00
A 48 month loan at 7.25% APR
Your monthly payment will be $377.78
Total cost will be $23633.43
```

You can see that for the same amount if you borrow it for a longer period you can reduce your monthly payments by over \$100 (but you pay about \$700 more in the end).

Decision Making in Computer Programs

Decision making is central to all computer programs. In fact there is a famous theorem in computer science that says that if any programmable device is capable of being programmed to do *sequential execution*, *decision making*, and *repetition*, it is capable of expressing any computable algorithm. This is a very powerful idea that is couched in terms of very simple ideas. Given a problem, if you can describe its solution in terms of a combination of the three execution models then you can get any computer to solve that problem. In this section, we will look at decision making in a classic game playing situation.

Computers have been extensively used in game playing: for playing games against people and other computers. The impact of computers on game playing is so tremendous that these days several manufacturers make game playing consoles with computers in them that are completely dedicated to game playing.

Let us design a simple game that you have most likely played while growing up: Paper, Scissors, Rock!

In this game, two players play against each other. At the count of three each player makes a hand gesture indicating that they have selected one of the three items: paper (the hand is held out flat), scissors (two fingers are extended to designate scissors, or rock (indicated by making a fist). The rules of deciding the winner are simple: if both players pick the same object it is a draw; otherwise, paper beats rock, scissors beat paper, and rock beats scissors. Let us write a computer program to play this game against the computer. Here is an outline for playing this game once:

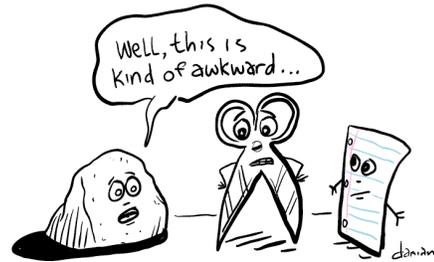
```
// Computer and player make a selection
// Decide outcome of selections (draw or who won)
// Inform player of decision
```

If we represent the three items in a string array, we can have the computer pick one of them at random by using the random number generation facilities provided in C++. If the items are represented as:

```
string items[] = {"Paper", "Scissors", "Rock"};
```

Then we can select any of the items above as the computer's choice using the expression:

```
// Computer makes a selection
string myChoice = items[<0 or 1 or 2 selected randomly>];
```



The exact name of the game can vary, with the three components appearing in a different order, or with “stone” in place of “rock”. Non-English speakers may know the game by their local words for “rock, paper, scissors”, although it is also known as *Jankenpon* in Japan, *Rochambeau* in France, and in South Africa as *Ching-Chong-Cha...*

From: talkingtotan.wordpress.com/2007/08

That is `items[0]` represents the choice "Paper", `items[1]` represents "Scissors", and `items[2]` is "Rock". We saw, in Chapter 4, that we can generate a random number in any range using the `rand` function from the `cstdlib` library in C++. Thus we can model the computer making a random selection using the following statement:

```
#include <cstdlib>
// Computer makes a selection
string myChoice = items[rand() % 3];
```

Recall that `rand() % n` returns a random number in the range $[0..n-1]$. Thus, `rand() % 3` will return either 0, 1, or 2. We can use the `askQuestion` command in Myro to ask the player to indicate their selection (see Chapter 3).

```
string choices = "Paper, Scissors, Rock";
// Player makes a selection
string yourChoice = askQuestion("Pick an item.", choices);
```

Now that we know how to the computer and player make their selection, we need to think about deciding the outcome. Here is an outline:

```
if both picked the same item then it is a draw
if computer wins then inform the player
if player wins then inform the player
```

Rewriting the above using `if`-statements we get the following first draft:

```
if (myChoice == yourChoice) {
    cout << "It is a draw.\n";
} else if (<myChoice beats yourChoice>) {
    cout << "I win.\n";
} else {
    cout << "You win.\n";
}
```

All we need to do is figure out how to write the condition `<myChoice beats yourChoice>`. The condition has to capture the rules of the game mentioned above. We can encode all the rules in a conditional expression as follows:

```
if ((myChoice == "Paper" && yourChoice == "Rock")
    || (myChoice == "Scissors" && yourChoice == "Paper")
    || (myChoice == "Rock" && yourChoice == "Scissors")) {
    cout << "I win.\n";
} else {
    cout << "You win.\n";
}
```

The conditional expression above captures all of the possibilities that should be examined in order to make the decision (recall that `&&` means “and” and `||` means “or”). Another way of writing the above decision would be to use the following:

```
if (myChoice == "Paper" && yourChoice == "Rock") {
    cout << "I win.\n";
} else if (myChoice == "Scissors" && yourChoice == "Paper") {
    cout << "I win.\n";
} else if (myChoice == "Rock" && yourChoice == "Scissors") {
    cout << "I win.\n";
} else {
    cout << "You win.\n";
}
```

That is each condition is examined in turn until one is found that confirms that the computer wins. If none such condition is true, the `else`-part of the `if`-statement will be reached to indicate that the player won.

Another alternative to writing the decision above is to encode the decision in a function. Let us assume that there is a function `beats` that returns `true` or `false` depending on the choices. We could then rewrite the above as follows:

```
if (myChoice == yourChoice) {
    cout << "It is a draw.\n";
} else if (beats(myChoice, yourChoice)) {
    cout << "I win.\n";
} else {
    cout << "You win.\n";
}
```

Let us take a closer look at how we could define the `beats` function. It needs to return `true` if `myChoice` beats `yourChoice`. So all we need to do is encode the rules of the game described above. Here is a draft of the function:

```
bool beats (string me, string you) {
    // Does me beat you? If so, return true, false otherwise.

    if (me == "Paper" && you == "Rock") {
        // Paper beats rock
        return true;
    } else if (me == "Scissors" && you == "Paper") {
        // Scissors beat paper
        return true;
    } else if (me == "Rock" && you == "Scissors") {
        // Rock beats scissors
        return true;
    } else {
        return false;
    }
}
```

Once again, we have used the `if`-statements in C++ to encode the rules of the game. Now that we have completely fleshed out all the critical parts of the program, we can put them all together as shown below:

```
// A program that plays the game of Paper, Scissors, Rock!
#include <iostream>
#include <string>
#include <cstdlib>
#include "Myro.h"
using namespace std;

bool beats (string me, string you) {
    // Does me beat you? If so, return true, false otherwise.

    if (me == "Paper" && you == "Rock") {
        // Paper beats rock
        return true;
    } else if (me == "Scissors" && you == "Paper") {
        // Scissors beat paper
        return true;
    }
}
```

```
    } else if (me == "Rock" && you == "Scissors") {
        // Rock beats scissors
        return true;
    } else {
        return false;
    }
}

int main() {
    string items[] = {"Paper", "Scissors", "Rock"};
    string choices = "Paper,Scissors,Rock";

    // Play a round of Paper, Scissors, Rock!
    cout << "Lets play Paper, Scissors, Rock!\n";
    cout <<
        "In the window that pops up, make your selection>\n";

    // Computer and Player make their selection...
    // Computer makes a selection
    string myChoice = items[rand() % 3];

    // Player makes a selection
    string yourChoice = askQuestion("Pick an item.", choices);

    // inform Player of choices
    cout << endl;
    cout << "I picked " << myChoice << endl;
    cout << "You picked " << yourChoice << endl;

    // Decide if it is a draw or a win for someone
    if (myChoice == yourChoice) {
        cout << "We both picked the same thing.\n";
        cout << "It is a draw.\n";
    } else if (beats(myChoice, yourChoice)) {
        cout << "Since " << myChoice << " beats "
            << yourChoice << "... \n";
        cout << "I win.\n";
    } else {
        cout << "Since " << yourChoice << " beats "
            << myChoice << "... \n";
        cout << "You win.\n";
    }
}
```

```
    cout << "Thank you for playing. Bye!\n";  
    return 0;  
}
```

A few more output commands were added to make the interaction more natural.

Do This: Implement the Paper, Scissors, Rock program from above and play it several times to make sure you understand it completely. Modify the program above to play several rounds. Also, incorporate a scoring system that keeps track of the number of times each player won and also the number of draws.

Summary

In this chapter you have seen a variety of control paradigms: behavior-based control for robots, writing a simple, yet useful computational program, and writing a simple computer game. Behavior-based control is a powerful and effective way of describing sophisticated robot control programs. This is the paradigm used in many commercial robot applications. You should try out some of the exercises suggested below to get comfortable with this control technique. The other two programs illustrate how, using the concepts you have learned, you can design other useful computer applications. In the next several chapters, we will explore several other dimensions of computing: media computation, writing games, etc.

Myro Review

There was nothing new from Myro in this chapter.

C++ Review

```
struct <structure name> { <declaration>; <declaration>; ... };
```

Defines a structure called `<structure name>` with *members* declared by the `<declarations>`s (usually variable and function declarations).

```
<structure name> <variable name>, <variable name>, ...;
```

Defines variables belonging to a structured type. Each variable has the members declared in the preceding `<structure name>` definition.

```
<structure>.<variable name>  
<structure>.<function name> ( <arguments> )
```

Accesses named member-variable (field) or member-function of a `<structure>`, which must have a structured type that declares the referenced member variable or function.

```
printf (<format string>, <expression>, ... );
```

Prints the `<expression>`s according to the `<format string>`. Within it, `%W.Df` prints a floating-point number in a field of minimum width `W` with `D` digits after the decimal point, `%Wd` prints an integer in a field of minimum width `W`, and `%Ws` prints a string in a field of minimum width `W`. The value is right-justified in the field, unless a minus sign follows the `%` (e.g., `%-W.Df`, `%-Wd`, `%-Ws`), in which case it's left-justified. There are many other format specifications besides these. In a `<format string>`, `%%` prints a single percent sign.

The math library module provides several useful mathematics functions. Some of the commonly used functions are listed below:

ceil(x) Returns the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

floor(x) Returns the floor of x as a float, the largest integer value less than or equal to x.

exp(x) Returns e^x .

log(x[, base]) Returns the logarithm of x to the given base. If the base is not specified, return the natural logarithm of x (i.e., $\log_e x$).

log10(x) Returns the base-10 logarithm of x (i.e. $\log_{10} x$).

pow(x, y) Returns x^y .

sqrt(x) Returns the square root of x (\sqrt{x}).

Trigonometric functions

acos(x) Returns the arc cosine of x, in radians.

asin(x) Returns the arc sine of x, in radians.

atan(x) Returns the arc tangent of x, in radians.

cos(x) Returns the cosine of x radians.

sin(x) Returns the sine of x radians.

tan(x) Returns the tangent of x radians.

cosh(x) Returns the hyperbolic cosine of x.

sinh(x) Returns the hyperbolic sine of x.

tanh(x) Returns the hyperbolic tangent of x.

Exercises

1. Replace the `avoid` module in the behavior-based control program with a module called: `follow`. The `follow` module tries to detect a wall to the robot's right (or left) and then tries to follow it at all times. Observe the resulting behavior.
2. Do population growth by modeling it as a difference equation.
3. Do population growth by modeling it as a continuous growth function.
4. Can you make the Paper, Scissors, Rock program better?
5. Implement the HiLo game: the computer picks a random number between 0 and 1000 and the user has to try and guess it. With every guess the computer informs the user of their guess was high (Hi), low (Lo), or they got it.
6. Reverse the roles in the HiLo game from above. This time, the user guesses a number between 0 and 1000 and computer program has to guess it. Think about a good strategy for guessing the number based on high and low clues, and then implement that strategy.

