11. Enumeration constants (and a longer example)

Herman Kamper

This lecture does two things. The first is small and quick: a way of inventing your own type, so that a variable can hold a value called `LEFT` or `ROCK` instead of a bare number that you then have to remember the meaning of. The second is bigger. We are going to sit down and write a complete program from nothing: a game of rock, paper, scissors that you can actually play against the computer.

The second half is really the point. Everything in it you have already met separately: variables, `if`, `switch`, functions, `rand`. What you have not seen yet is the whole process, from a blank file to a working program, including the part where you stop and think before you type. That part turns out to matter more than the syntax.

A robot that keeps turning the wrong way

Suppose you are writing a program that drives a robot around. At any moment the robot is facing in one of four directions: up, down, left or right. You need to store that direction somewhere, so you need a variable, and the variable needs a type.

The obvious thing to reach for is `int`, and then to decide on an encoding: 0 means up, 1 means down, 2 means left and 3 means right.

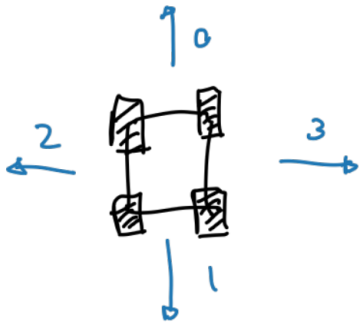

Figure 1: Four directions, encoded as the integers 0 to 3.

So you write something like this:

```
int robot1Direction;
int robot2Direction;

robot1Direction = 2;
```

This works. It compiles, it runs, and the robot goes left. But read that last line again, cold, as if you had not just written it. What does 2 mean? You have to go and look it up, or remember it. And I am terrible at this. I struggle with left and right in real life, never mind when they have been dressed up as 2 and 3.

Now imagine a hundred lines further down:

```
if (robot1Direction == 2)
    printf("You are moving left.\n");
```

Nothing here says "left" anywhere. The number carries the meaning, but only in your head, and heads are unreliable. What you actually want is to invent a new type of your own, called something like `Direction`, whose values are `UP`, `DOWN`, `LEFT` and `RIGHT`. Then the program says what it means.

That is what enumeration constants give you.

Defining your own type

Here is the whole idea in one line:

```
enum Direction {UP, DOWN, LEFT, RIGHT};
```

You write the keyword `enum`, then a name for your new type, then a list of the values it is allowed to take, in curly braces, separated by commas. That is it. You have just created a new kind of thing.

It is worth pausing on how big a deal that is. Up to now the types available to you were the ones C came with: `int`, `double`, `float`, `char`. Those were fixed. You could not add to the list. Now you can. In the same way that writing a function gives you a new command that did not exist before, writing an `enum` gives you a new type that did not exist before. The language grows to fit the problem you are solving, rather than the other way round.

Once the type exists you can declare variables of that type:

```
enum Direction robot1Direction;
```

Note that the type is `enum Direction`, both words, not just `Direction`. This catches everyone out, and it catches me out every single time. When you define the type you write `enum`, and when you declare a variable of that type you write `enum` again. If I were designing C from scratch I would not have done it this way. There is a way to get rid of the second `enum` using something called `typedef`, but you do not need it yet.

Putting it together:

```c
#include <stdio.h>

enum Direction {UP, DOWN, LEFT, RIGHT};

int main()
{
    enum Direction robot1Direction;

    robot1Direction = LEFT;

    if (robot1Direction == LEFT)
    {
        printf("You are moving left.\n");
    }
    else if (robot1Direction == UP)
    {
        printf("You are moving up.\n");
    }

    return 0;
}
```

Run it and you get:

```
You are moving left.
```

Change the assignment to `robot1Direction = UP;` and you get "You are moving up." instead. The behaviour is identical to the version with the numbers. The difference is entirely in what happens when a human reads it, which is a difference worth a great deal.

What is really going on underneath

Internally, C does not do anything clever here. It just turns your names into integers, counting from zero in the order you wrote them:

Constant	Value
`UP`	0
`DOWN`	1
`LEFT`	2
`RIGHT`	3

So `LEFT` really is 2, and `robot1Direction == LEFT` really is `robot1Direction == 2`. Nothing has changed about what the machine does. What has changed is that you no longer have to hold the table in your head, because you wrote it down once, at the top of the file, in a place the compiler can read.

Two consequences follow from enumeration constants being integers underneath.

The first is that you cannot read one directly from the user. `scanf` with `%d` promises to write an `int` into the box you give it, and an enumeration variable is not guaranteed to be exactly that. So read an `int` first, then assign it across:

```
int shapeInput;
enum Shape playerShape;

scanf("%d", &shapeInput);
playerShape = shapeInput;
```

The second is that the compiler will not stop you assigning a nonsense value. Nothing prevents `robot1Direction = 47;`. An enumeration is a set of convenient names, not a guarantee, and checking that the input makes sense is still your job.

One last small thing: the capital letters are a convention, not a rule. `enum Direction {up, down, left, right};` compiles perfectly well. But writing them in capitals is a signal to anyone reading that these are fixed, named values rather than variables, and it is worth following.

A longer example: rock, paper, scissors

Now for the real work of this lecture. We are going to write a complete program: you pick rock, paper or scissors, the computer picks one at random, and the program tells you who won.

The rules, in case they need stating: each player picks one of three shapes. Rock beats scissors, scissors beats paper, and paper beats rock. If both players pick the same shape it is a draw.

You know the rules. You know all the C you need. The question is what you do first, and the answer is not "start typing".

Step one: work out what should happen

Before writing a single line of code, work out what the program is supposed to do. For a problem this small it feels like overkill, and honestly it is. But the habit is what matters, because your programs will not stay this small, and the moment they get complicated is exactly the moment you will not think to stop and do it.

Here it means two things. First, being clear about the shape of the program: the player chooses, the computer chooses at random, we work out who won, we say so. Four steps.

Second, being clear about the rules themselves, which is best done by drawing them. Three shapes, arrows for who beats whom, loops back on themselves for the draws.

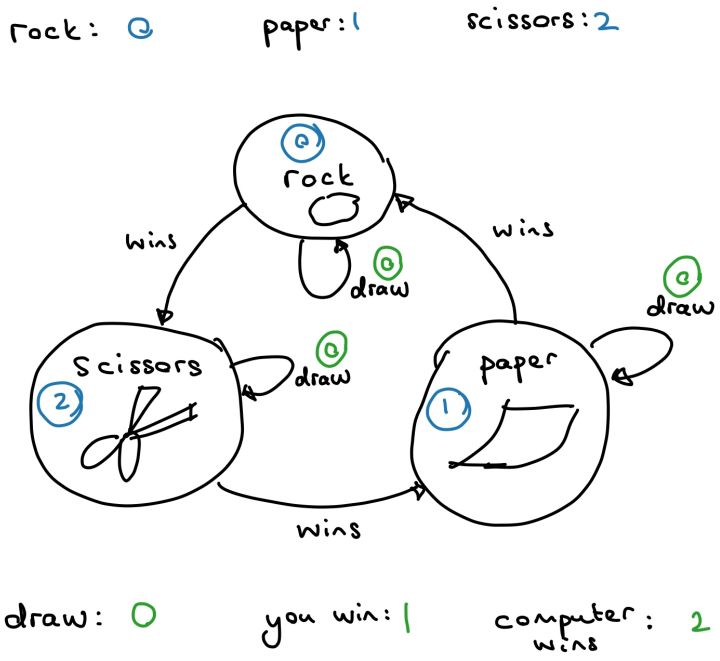

Figure 2: The rules of the game, drawn out. The arrows point from the winning shape to the shape it beats, and the loops are the draws. The blue numbers are the encoding for the shapes and the green ones for the outcomes.

That picture is worth more than it looks. Once it exists, you have solved the problem: everything left is translation. Whenever you find yourself confused halfway through writing something,

it is usually because you skipped this and started coding a problem you had not finished understanding.

While drawing it, we also have to decide how the shapes will be represented. For now, as integers: rock is 0, paper is 1, scissors is 2. (Rock is 0 because a rock is round and so is a zero. I need help of this quality to remember anything.)

We need a second encoding too, for the result of a round: 0 for a draw, 1 for the player winning, 2 for the computer winning.

Keep an eye on those two encodings. They are both 0, 1, 2, and they mean completely different things. This is going to matter shortly.

Step two: what pieces will you need?

Take the four steps and ask, for each one, what C you will use.

Reading the player's shape is `scanf`. Generating the computer's shape is `rand`, and since we want a shape rather than an enormous number, `rand() % 3`, which gives 0, 1 or 2. Three shapes, so modulus 3. And `rand` needs `srand` alongside it, which does not give you a random number itself but sets the starting point for the ones that follow.

Printing the result is `printf`, with a `switch` in front of it to pick which of the three messages to print.

That leaves the interesting step: working out who won. What do we use for that? The honest answer is that we do not want to think about it yet. So we write a function, give it a name, decide what goes in and what comes out, and then leave it alone:

```
int getRoundStatus(int playerShape, int computerShape);
```

Two shapes go in, a round status comes out. Deciding this and moving on is the single most useful thing functions let you do. The hard part of the problem is now behind a name, and we can build the rest of the program around it as though it were already finished.

Here is the skeleton, with the awkward bits left as comments:

```c
#include <stdio.h>
#include <stdlib.h>

int getRoundStatus(int playerShape, int computerShape);

int main(void)
{
    int playerShape, computerShape, roundStatus;

    // Read the player's choice (scanf)
    playerShape = 0;      // rock, for now

    // Read the computer's choice (srand, rand)
    computerShape = 1;    // paper, for now

    // Check who won
    roundStatus = getRoundStatus(playerShape, computerShape);

    // Print who won
    switch (roundStatus)
    {
        case 0:
            printf("Round drawn.\n");
            break;

        case 1:
            printf("You won!\n");
            break;

        case 2:
            printf("Computer won!\n");
            break;
    }

    return 0;
}
```

Notice that the two choices are hard-coded for the moment. This is deliberate and it is worth copying. A program that always plays the same round is a program you can test, because you know what the answer should be. Randomness is the last thing you want while you are still finding out whether your logic works.

Step three: working out who won

Now the function we postponed. It receives the two shapes and has to return 0, 1 or 2.

Start with the easy case. If the two shapes are the same, it is a draw, and we are done:

```
if (playerShape == computerShape)
{
    return 0;       // draw
}
```

That `return` sitting in the middle of a function is worth a comment, because it is the first time we have used one this way. The moment `return` executes, the function is over. Not the `if`, not the block: the whole function. Control jumps straight back to whoever called it, carrying the value with it, and nothing below runs at all. So there is no need for an `else` here. If it was a draw we have already left.

For everything else, we work through the three things the player could have chosen, and in each case ask whether the computer picked the shape that loses to it:

```
switch (playerShape)
{
    case 0:                             // player did rock
        if (computerShape == 2)         // computer did scissors
        {
            return 1;                   // player wins
        }
        break;

    case 1:                             // player did paper
        if (computerShape == 0)         // computer did rock
        {
            return 1;
        }
        break;

    case 2:                             // player did scissors
        if (computerShape == 1)         // computer did paper
        {
            return 1;
        }
        break;
}

return 2;       // otherwise the computer won
```

The strategy is: check the three ways the player can win, and if none of them happened, then, given that we have already ruled out a draw, the player must have lost. So the function ends

with `return 2` and every path that does not win falls through to it.

That last part is the bit people trip over, so let us walk one round through it slowly. Say you play rock and the computer plays paper.

We arrive at the top of the function. Are the shapes equal? Rock is 0 and paper is 1, so no, and we skip the draw. We reach the `switch` and jump to `case 0`, because you played rock. Inside it we ask whether the computer played scissors. It played paper, so the condition is false and the `if` body does not run. We hit `break`, which takes us out of the whole `switch`. And immediately below the `switch` sits `return 2`, which is a loss, which is exactly right: paper covers rock.

So the `break` is not decoration. It is what carries a non-winning case down to the loss at the bottom.

Testing it as you go

I write one line at a time and run the program after each one. This sounds excessive and it is the single biggest thing that separates a pleasant afternoon from a miserable one. If you write forty lines and then compile, and it is wrong, the mistake is somewhere in forty lines. If you write one line and run it, the mistake is in that line.

The same applies to the hard-coded shapes. With the computer forced to play scissors, playing rock should print "You won!" and playing paper should print "Computer won!" Those are two facts you can check by hand. Change the forced shape to rock and check three more. By the time you unleash `rand` on it you already know the logic is right, and if the finished program then misbehaves you know where not to look.

The problem with all of this

The program works. Play a few rounds and you will find it gives the right answers.

But try reading it. Here is a line from the middle of the function:

```
case 1:
    if (computerShape == 0)
```

What does that say? It says: if the player chose paper and the computer chose rock. But you cannot see that. You have to remember two encodings at once, both of which run 0, 1, 2, and neither of which is written down anywhere in the code. And it gets worse, because `return 1` a couple of lines later is the other encoding, the one where 1 means the player won.

By this point in the lecture I had thoroughly confused myself while writing it out, which was not the plan but did make the point better than I could have. The program is correct and nearly unreadable, and correct-but-unreadable is a temporary state. It becomes incorrect the moment somebody changes it.

This is the problem enumeration constants solve.

The same program, with enums

We introduce a type for the shapes:

```
enum Shape {ROCK, PAPER, SCISSORS};
```

Because the constants take the values 0, 1 and 2 in that order, this lines up exactly with the encoding we were already using. Nothing about the running program changes. The variables become `enum Shape` instead of `int`, in the declarations and in the function prototype, and every bare number gets a name.

Here is the function again:

```
int getRoundStatus(enum Shape playerShape, enum Shape computerShape)
{
    if (playerShape == computerShape)
    {
        return 0;
    }

    switch (playerShape)
    {
        case ROCK:
            if (computerShape == SCISSORS)
            {
                return 1;
            }
            break;

        case PAPER:
            if (computerShape == ROCK)
            {
                return 1;
            }
            break;

        case SCISSORS:
            if (computerShape == PAPER)
            {
                return 1;
            }
            break;
    }

    return 2;
}
```

Compare `case ROCK:` with `case 0:`. The logic is character-for-character the same and the

compiled program is identical. But now the code states the rules of the game instead of encoding them, and you can check it against the diagram just by reading it: rock beats scissors, paper beats rock, scissors beats paper. Three lines, three rules.

The return values are still bare numbers, though, which is the half of the confusion we have not fixed yet.

An enum for the outcome too

The same trick applies to the result of a round:

```
enum Outcome {DRAW, WIN, LOSS};
```

The order matters here, and it is not arbitrary. We chose it so that `DRAW`, `WIN` and `LOSS` come out as 0, 1 and 2, matching the encoding the program already used. Now `getRoundStatus` can return an `enum Outcome` rather than an `int`, and every `return 0` becomes `return DRAW`, every `return 1` becomes `return WIN`, and the fall-through at the bottom becomes `return LOSS`.

Here is the finished program.

```
/*
 * A rock, paper, scissors game.
 * Author: Herman Kamper
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum Shape {ROCK, PAPER, SCISSORS};
enum Outcome {DRAW, WIN, LOSS};

enum Outcome getRoundStatus(enum Shape playerShape,
                            enum Shape computerShape);

int main(void)
{
    enum Shape playerShape, computerShape;
    enum Outcome roundStatus;
    int shapeInput;

    // Read in the player's shape
    printf("Enter your shape (0: rock, 1: paper, 2: scissors): ");
    scanf("%d", &shapeInput);
    playerShape = shapeInput;

    // Generate a random shape for the computer
```

```
    srand(time(NULL));
    computerShape = rand() % 3;

    switch (computerShape)
    {
        case ROCK:
            printf("Computer chose rock.\n");
            break;

        case PAPER:
            printf("Computer chose paper.\n");
            break;

        case SCISSORS:
            printf("Computer chose scissors.\n");
            break;
    }

    // Check who won
    roundStatus = getRoundStatus(playerShape, computerShape);

    // Print who won
    switch (roundStatus)
    {
        case DRAW:
            printf("Round drawn.\n");
            break;

        case WIN:
            printf("You won!\n");
            break;

        case LOSS:
            printf("Computer won!\n");
            break;
    }

    return 0;
}


/*
 * Returns DRAW, WIN or LOSS from the player's point of view.
 */
enum Outcome getRoundStatus(enum Shape playerShape,
```

```
                                enum Shape computerShape)
{
    if (playerShape == computerShape)
    {
        return DRAW;
    }

    switch (playerShape)
    {
        case ROCK:
            if (computerShape == SCISSORS)
            {
                return WIN;
            }
            break;

        case PAPER:
            if (computerShape == ROCK)
            {
                return WIN;
            }
            break;

        case SCISSORS:
            if (computerShape == PAPER)
            {
                return WIN;
            }
            break;
    }

    return LOSS;
}
```

Playing a round looks like this:

```
Enter your shape (0: rock, 1: paper, 2: scissors): 0
Computer chose scissors.
You won!
```

Note that `srand(time(NULL))` is back in, so the computer's choice now genuinely varies from run to run. Take it out and the computer will play the same sequence of shapes every time you start the program, which is a disaster in a finished game and rather useful while you are still testing one.

A neater way to do the same thing

There is a much shorter way to write `getRoundStatus`, which a student pointed out at the end of the lecture and which is better than mine. Instead of a `switch` with three cases each containing an `if`, just ask the whole question at once:

```
enum Outcome getRoundStatus(enum Shape playerShape,
                            enum Shape computerShape)
{
    if (playerShape == computerShape)
    {
        return DRAW;
    }

    if ((playerShape == ROCK && computerShape == SCISSORS)
        || (playerShape == PAPER && computerShape == ROCK)
        || (playerShape == SCISSORS && computerShape == PAPER))
    {
        return WIN;
    }

    return LOSS;
}
```

Three conditions joined with `||`, one for each way the player can win. It gives exactly the same answer for all nine combinations, it is a third of the length, and the three winning rules sit next to each other where you can see them all at once. If you were writing this out by hand, this is the version to write.

I showed you the `switch` version first partly because it is a good exercise in `switch` and `break`, and partly because the long version is genuinely how a first attempt tends to look. Getting something working and then noticing it could be half the size is a completely normal way to arrive at good code.

Looking forward

Enumeration constants are a small feature. You could write every program in this course without them and nothing would break. But they are your first taste of something that runs through the rest of programming: shaping the language around the problem, so that the code describes what you meant rather than merely achieving it.

The bigger lesson is the process. Draw the thing. Decide what the pieces are. Hide the hard part behind a function name and build around it. Hard-code the inputs so you can test. Write one line at a time. None of that is C, and all of it will still be true in whatever language you are using in ten years.

The next lecture returns to functions from an unexpected angle: what happens when a function calls itself. That is recursion, and it is one of the genuinely surprising ideas in programming. To

make sense of it we will also need to answer a question we have been stepping around: when a function creates a variable, where does that variable actually live, who else can see it, and how long does it survive? Those are the scope and storage class rules, and they are what make recursion possible.

Exercises

1. Write a program that stores a direction using `enum Direction {UP, DOWN, LEFT, RIGHT};` and uses a `switch` to print a sentence describing it. Then print the direction with `printf("%d\n", direction);` and confirm that you get the number you expect.
2. What does `enum Grade {A = 5, B, C, D};` give you for each constant? The rule is that a constant with no value assigned is one more than the one before it. Work it out on paper, then write a short program to check.
3. Take the finished game and rewrite `getRoundStatus` using the `||` version above. Verify by hand that both versions agree on all nine combinations of shapes.
4. The game currently plays a single round. Wrap it in a loop so that it keeps playing until the player enters something outside 0 to 2, and print the running score at the end.
5. At the moment, entering 7 as your shape produces nonsense rather than a complaint. Add a check that rejects anything outside the valid range and asks again.
6. Extend the game to rock, paper, scissors, lizard, Spock. Lizard beats paper and Spock, Spock beats rock and scissors, and the original three rules stand. Add the two constants to `enum Shape`, and think about which of the two versions of `getRoundStatus` you would rather be extending. That answer alone is a decent argument for the shorter one.
7. Write a program that plays the computer against itself for 1000 rounds, with both shapes chosen by `rand`, and counts draws, wins and losses. Roughly what proportion would you expect for each before running it? Does the program agree?