Improved Code Quality

This lesson explains how the use of functions improves the code quality.

WE'LL COVER THE FOLLOWING

Code quality through functions

Functions can improve the quality of code. Smaller functions with fewer responsibilities lead to programs that are easier to read and maintain.

Code duplication is harmful

One aspect that is highly detrimental to program quality is code duplication. **Code duplication** occurs when there is more than one piece of code in the program that performs the same task.
Although this is sometimes done intentionally by copying lines of code around, it may also happen accidentally when writing separate pieces of code.

One of the problems with pieces of code that duplicate essentially the same function is that they present multiple chances for bugs to crop up. It can be hard to make sure that we have fixed all the places where we introduced the problem, as they may be spread around. Conversely, when the code appears in only one place in the program, then we only need to fix it at that one place to get rid of the bug once and for all.

Functions are closely related to the craft aspect of programming. We should always be on the lookout for code duplication and continually try to identify commonalities in code that could be moved to individual functions.

Example

Let’s start with a program that contains some code duplication. Let’s see how that duplication can be removed by moving code into functions. The following program reads numbers from the input. It first displays them in the order that they have been read and then displays them in sorted numerical order:

```
import std.stdio;
import std.algorithm;

void main() {
    int[] numbers;

    int count;
    write("How many numbers are you going to enter? ");
    readf(" %s", &count);

    // Read the numbers
    foreach (i; 0 .. count) {
        int number;
        write("Number ", i, "? ");
        readf(" %s", &number);

        numbers ~= number;
    }

    // Print the numbers
    writeln("Before sorting:");
    foreach (i, number; numbers) {
        writefln("%3d:%5d", i, number);
    }

    sort(numbers);

    // Print the numbers
    writeln("After sorting:");
    foreach (i, number; numbers) {
        writefln("%3d:%5d", i, number);
    }
}
```

Program with code duplication

Some of the duplicated lines of code are obvious in that program. The last two `foreach` loops that are used for displaying the numbers are exactly the same.

Defining a function that might appropriately be named as `display()` would remove that duplication. The function could take a slice as a parameter and print it:

```
void display(int[] slice) {
    foreach (i, element; slice) {
        writefln("%3s:%5s", i, element);

    }
}
```

Notice that the parameter now uses a more general name `slice` instead of the original and more specific name `numbers`. The reason is that the function does not need to know what the elements of the slice would specifically represent. This information is only available at the time and place the function is called. The elements may be student IDs, parts of a password, etc. `slice` and `element` are more generic names and can be used to represent multiple types. The new function can be called from the two places where the slice needs to be displayed:

```
import std.stdio;
import std.algorithm;

void display(int[] slice) {
    foreach (i, element; slice) {
        writefln("%3s:%5s", i, element);
    }
}

void main() {
    int[] numbers;

    int count;
    write("How many numbers are you going to enter? ");
    readf(" %s", &count);

    // Read the numbers
    foreach (i; 0 .. count) {
        int number;
        write("Number ", i + 1, "? ");
        readf(" %s", &number);

        numbers ~= number;
    }

    // Print the numbers
    writeln("Before sorting:");
    display(numbers);

    sort(numbers);

    // Print the numbers
    writeln("After sorting:");
    display(numbers);
}
```

Code with the print() function

There is more to do. Notice the title line is written just before displaying the elements of the slice. Although both title lines are different, the task of displaying them is the same. Displaying the title can be included in the function, and the title too can be passed as a parameter. Here are the new changes:

```
void display(string title, int[] slice) {
    writeln(title, ":");

    foreach (i, element; slice) {
        writefln("%3s:%5s", i, element);
    }
}
// ...
  // display the numbers
    display("Before sorting", numbers);
// ...
  // display the numbers
    display("After sorting", numbers);
```

This step has the added benefit of obviating the comments that appear right before the two `display()` calls. Since the name of the function already clearly communicates what it does, those comments are unnecessary:

```
display("Before sorting", numbers);
sort(numbers);
display("After sorting", numbers);
```

Although subtle, there is more code duplication in this program. The values of `count` and `number` are read in exactly the same way. The only difference is the message that is printed to the user and the name of the variable:

```
int count;
write("How many numbers are you going to enter? ");
readf(" %s", &count);

// ...

int number;
write("Number ", i, "? "); readf(" %s", &number);
```

The code would become even better if it took advantage of a new function that might be named appropriately as `readInt()`. The new function can take the message as a parameter, print that message, read an `int` from the input and return that `int`:

```
int readInt(string message) {
    int result;
    write(message, "? ");
    readf(" %s", &result);
    return result;
}
```

`count` can now be initialized directly by the return value of a call to this new function:

```
int count = readInt("How many numbers are you going to enter");
```

`number` cannot be initialized in as straightforward a way because the loop counter `i` happens to be a part of the message that is displayed when reading number. This can be overcome by taking advantage of the `format()`:

```
import std.string;
// ...
    int number = readInt(format("Number %s", i));
```

Furthermore, since number is used in only one place in the `foreach` loop, its definition can be eliminated altogether and the return value of `readInt()` can directly be used in its place:

```
foreach (i; 0 .. count) {
    numbers ~= readInt(format("Number %s", i));
}
```

Let’s make a final modification to this program by moving the lines that read the numbers to a separate function. This would also eliminate the need for the “read the numbers” comment because the name of the new function would already carry that information.

The new `readNumbers()` function does not need any parameter to complete its task. It reads some numbers and returns them as a slice. The following is the final version of the program:

```
import std.stdio;
import std.string;
import std.algorithm;

void display(string title, int[] slice) {
    writeln(title, ":");

    foreach (i, element; slice) {
        writefln("%3s:%5s", i, element);
    }
}

int readInt(string message) {
    int result;
    write(message, "? ");
    readf(" %s", &result);
    return result;
}

int[] readNumbers() {
    int[] result;

    int count =
        readInt("How many numbers are you going to enter");

    foreach (i; 0 .. count) {
        result ~= readInt(format("Number %s", i));
    }

    return result;
}

void main() {
    int[] numbers = readNumbers();
    display("Before sorting", numbers);
    sort(numbers);
    display("After sorting", numbers);
}
```

Final version of the program

Compare this version of the program to the first one. The major steps of the program are very clear in the `main()` function of the new program. In contrast, the `main()` function of the first program had to be carefully examined to understand the purpose of that program.

Although the total number of non-trivial lines of the two versions of the program end up being equal in this example, functions generally make programs shorter. This effect is not apparent in this simple program. For example, before the `readInt()` function has been defined, reading an `int` from the input involved three lines of code. After the definition of `readInt()`, the same goal is achieved by a single line of code. Furthermore, the definition

of `readInt()` allowed the removal of the definition of the variable `number` altogether.

Commented lines of code as functions

Sometimes the need to write a comment that describes the purpose of a group of lines of code is an indication that those lines could better be moved to a newly defined function. If the name of the function is descriptive enough, there will be no need for the comment either.
The three commented groups of lines of the first version of the program have been used for defining new functions that achieved the same tasks.

In the next lesson, you will find a coding challenge based on the use of functions.