Strings

This lesson explains strings, its types and different functions that can be performed on strings.

WE'LL COVER THE FOLLOWING

Strings

We have used strings in many of the programs that we have seen so far. Strings are a combination of the two features that we covered earlier in this chapter: *characters* and *arrays*. In the simplest definition, **strings** are nothing but arrays of characters. For example, `char[]` is a type of string.

This simple definition may be misleading. As we have seen in the characters: types and literals lesson, D has three separate character types. Arrays of these character types lead to three separate string types, some of which may have surprising outcomes in some string operations.

`readln` and `strip`, instead of `readf`

There are surprises even when reading strings from the terminal. Being character arrays, strings can contain control characters like `\n` as well. When reading strings from the input, the control character that corresponds to the enter key that is pressed at the end of the input becomes a part of the string as well. Further, because there is no way to tell `readf()` how many characters to read, it continues to read until the end of the entire input. For these reasons, `readf()` does not work as intended when reading strings:

```
char[] name;
write("What is your name? ");
readf(" %s", &name);
writeln("Hello ", name, "!");
```

The enter key that the user presses after the name does not terminate the input. `readf()` continues to wait for more characters to add to the string:

```
What is your name? Mert
← The input is not terminated although enter has been pressed
 ← (Let's assume that enter is pressed a second time here)
```

One way of terminating the standard input stream in a terminal is pressing Ctrl- D under Unix-based systems and Ctrl-Z under Windows systems. If the user eventually terminates the input that way, we see that the new-line characters have been read as parts of the string as well:

```
Hello Mert
← new-line character after the name
! ← (one more before the exclamation mark)
```

The exclamation mark appears after those characters instead of being printed right after the name.

`readln()` is more suitable when reading strings. Short for "read line," `readln()` reads until the end of the line. It is used differently because the `" %s"` format string and the `&` operator are not needed:

```
import std.stdio;

void main() {
    char[] name;

    write("What is your name? ");
    readf(" %s", &name);

    writeln("Hello ", name, "!");
}
```

readln() stores the new-line character as well. This is so the program has a way of determining whether the input consisted of a complete line or whether the end of input has been reached:

```
What is your name? Mert
Hello  Mert
! ← new-line character before the exclamation mark
```

Such control characters and all whitespace characters at both ends of strings can be removed by `std.string.strip:`

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

void main() {
    char[] name;

    write("What is your name? ");
    readln(name);
    name = strip(name);

    writeln("Hello ", name, "!");
}
```

Use of strip( )

The `strip()` expression above returns a new string that does not contain the trailing control characters. Assigning that return value back to name produces the intended output:

```
What is your name? Mert
Hello Mert! ← no new-line character
```

`readln()` can be used without a parameter. In that case it returns the line that it has just read. Chaining the result of `readln()` to `strip()` enables a shorter and more readable syntax:

```
string name = strip(readln());
```

`formattedread` for parsing strings

Once a line is read from the input or from any other source, it is possible to parse and convert separate data that it may contain with `formattedRead()` from the `std.format` module. Its first parameter is the line that contains the data, and the rest of the parameters are used exactly like `readf()`:

```
import std.stdio;
import std.string;
import std.format;

void main() {
    write("Please enter your name and age," ~
          " separated with a space: ");

    string line = strip(readln());

    string name;
    int age;
    formattedRead(line, " %s %s", name, age);

    writeln("Your name is ", name,
            ", and your age is ", age, '.');
}
```

Both `readf()` and `formattedRead()` return the number of items that they could parse and convert successfully. That value can be compared against the expected number of data items so that the input can be validated. For example, as the `formattedRead()` call above expects to read two items (a `string` as `name` and an `int` as `age`), the following check ensures that it really is the case:

```
uint items = formattedRead(line, " %s %s", name, age);

if (items != 2) {
  writeln("Error: Unexpected line.");

} else {
  writeln("Your name is ", name,", and your age is ", age, '.');
}
```

When the input cannot be converted to name and age, the program prints an error:

```
Please enter your name and age, separated with space: Mert
```

```
Error: Unexpected line.
```

Double quotes, not single quotes

We have seen that single quotes are used to define character literals. String literals are defined with double-quotes. *'a'* is a character; *"a"* is a string that contains a single character.

Immutability of `string`, `wstring` and `dstring`

There are three string types that correspond to the three character types:

- `char[]`
- `wchar[]`
- `dchar[]`

There are three aliases of the immutable versions of those types:

- `string`
- `wstring`
- `dstring`

The characters of the variables that are defined by these aliases cannot be modified. For example, the characters of a `wchar[]` can be modified, but the characters of a `wstring` cannot be modified. We will see D's immutability concept in later chapters.

For example, the following code that tries to capitalize the first letter of a string would cause a compilation error:

```
import std.stdio;

void main() {

    string cannotBeMutated = "hello";
    cannotBeMutated[0] = 'H'; // ← compilation ERROR

}
```

We may think of defining the variable as a `char[]` instead of the string alias, but that cannot be compiled either:

```
import std.stdio;

void main() {

    char[] a_slice = "hello"; // ← compilation ERROR
    a_slice[0] = 'H'; // ← compilation ERROR

}
```

This time the compilation error is due to the combination of two factors:

1. The type of string literals like “hello” is `string`, not `char[]`, so they are immutable.

2. The `char[]` on the left-hand side is a slice, which, if the code compiled, would provide access to all of the characters of the right-hand side.

Since `char[]` is mutable and string is not, there is a mismatch. The compiler does not allow accessing characters of an immutable array through a mutable slice. The solution here is to make a copy of the immutable string “hello” by using the `.dup` property:

```
import std.stdio;

void main() {
  char[] s = "hello".dup;

  s[0] = 'H';

  writeln(s);
}
```

Char and immutable string

The program can now be compiled and will print the modified string. Similarly, `char[]` cannot be used where a string is needed. In such cases, the `.idup` property can be used to produce an immutable string variable from a mutable `char[]` variable. For example, if `s` is a variable of type `char[]`, the following line will fail to compile:

```
import std.stdio;

void main() {
    char[] s = "hello".dup;

    string result = s ~ '.'; // ← compilation ERROR
}
```

When the type of `s` is `char[]`, the type of the expression on the right-hand side of the assignment above is `char[]` as well. `.idup` is used for producing immutable strings from existing strings:

```
import std.stdio;

void main() {
    char[] s = "hello".dup;

    string result = (s ~ '.').idup; // ← now compiles

    writeln(result);

}
```

`.icmp` for comparing strings

Because each character has a unique code, every letter variant is different from the others. For example, 'A' and 'a' are different letters when directly comparing Unicode strings.

Additionally, as a consequence of their ASCII code values, all of the Latin uppercase letters are sorted before all of the lowercase letters. For example, 'B' comes before 'a'. The `icmp()` function of the `std.string` module can be used when strings need to be compared regardless of lowercase and uppercase. You can see the functions of this module at its online documentation. Because strings are arrays (and like a corollary, ranges), the functions of the `std.array`, `std.algorithm` and `std.range` modules are very useful with strings as well.

In the next lesson, we will see string length, literals, and comparison.