# Unit 29: Standard I/O Functions

We have been shielding you from the pain and pitfalls of using C I/O functions. Since you are close to "graduate" from CS1010, this is a good time to show you what the real world is like.

## printf

The function `printf` is used to print a formatted string to the standard output. Unlike functions that we have seen so far, `printf` can take in a variable number of arguments, but it must have at least one argument -- a string written in a certain format. The rest of the arguments can be of different types. Let's look at an example:

```
1  char *name = "Siri";
2  printf("Hello! My name is %s.\n", name);
```

The code above prints `Hello! My name is Siri.`.

There are two arguments to `printf`, both are strings. The first is the string to print. There is a placeholder `%s` in the string, which will be replaced by the second argument `name`. The string to print ends with `\n`, which is the new line character.

The placeholder `%s` is called a *format modifier*. It controls how to interpret the arguments (i.e., what type) and how to format the output. The general format is:

```
1  %[flags][field_width][.precision][length_modifier]specifier
```

The letter after `%` controls the interpretation of the argument. `s` for string, `c` for character, `d` for integer (base 10), `f` for floating-point number, `p` for pointer (base 16). We can additionally prepend this with *length modifier*. `ld` for `long` integer, `lld` for `long long`, and `lf` for `double`.

To format the output, we can prepend it with a number to indicate its *field width*, or minimum space used when printing. E.g., `%3d` will pad the number printed with space if the number printed is less than 3 digits. Adding a *flag* 0 in front, `%03d`, will pad the number with 0s if the number printed is less than 3 digits. Other flags include `+`, which tells `printf` to print a sign (`+` or `-`) for the number. For floating-point numbers, we can additionally control the *precision*, or the number of digits printed after the decimal point. `%3.4lf` will print a double to four decimal points.

Note: `cs1010_println_double` uses `%.4lf` as the format modifier.

## Pitfalls when using `printf`

### Mismatch Types

`printf` does not check for the type of arguments we pass to it. The compiler does, but it only politely warns us instead of throwing an error like other type mismatches.

If you ignore such warnings, you might print strange things like:

```
1    printf("%d %f\n", 100000000000, 100000000000);
```

or worse, crash your program:

```
1    printf("%s %s\n", 100000000000, 100000000000);
```

### Mismatch Number of Arguments

Since `printf` expects a variable number of arguments, you can pass it fewer arguments than expected and the code would still compile (with warnings). If you push ahead and run it anyway, `printf` will start to fetch arguments from the stack, pretending that it is there, causing weird things to happen.

Consider:

```
1  printf("%d %s %s\n", 10);
```

It would cause `printf` to access the memory content of the stack as strings.

### Printing User Input

We should also never do this:

```
1  char *str = cs1010_read_word();
2  printf(str);
```

The reason is that we have no control over what the user would type as input: the user may type "%s" into the standard input, so the variable `str` now points to `%s`, which `printf` treats as a format modifier, and output the content of the stack! This is a huge security risk.

We should always print a string using:

```
printf("%s", str);
```

## scanf

The function `scanf` is used to read inputs from the standard input. It requires us to pass in pointers to variables where we want to store the input value is. Like `printf`, it takes in one or more arguments, with the first argument being a format string containing one or more format specifiers. The format specifier for `scanf` is simpler and has the following pattern:

```
%[*][field_width][length_modifier]specifier
```

For instance, to read an integer, a floating-point number, and a string of at most 10 characters,

```
long l;
double d;
char s[11];
scanf("%ld %lf %10s", &l, &d, s);
```

`scanf` scans the standard input, try to match it to the format specified. The space in between the format specifier matches zero or more white spaces (space, tab, newline). Scanning stops when an input character does not match such a format character or when an input conversion fails.

Adding a `*` to the format modifier means that `scanf` should consume the inputs but not store it in any variables. This, combined with `%[``]` is useful to clear any remaining data from the standard input.

## Pitfalls When Using scanf

### Checking for Error

The function `scanf` fails silently when the input character does not match a format or when the input conversion fails. It might return an unexpected input. We should always check the return value of `scanf` to make sure that it is reading properly.

```
long a;
scanf("%ld", &a);
printf("%ld", a);
```

The code above might print an uninitialized value if the input is not an integer.

We should check

```
long a;
long result = scanf("%ld", &a);
if (result == 1) {
  printf("%ld", a);
}
```

The above, however, does not properly "clear" the standard input of the incorrect input. So the next `scanf` calls would still try to read it again!

To clear the input, we can use the `%*[^\n]` modifier, which read in any characters expect (`^`) the newline (`\n`).

```
long a;
long result = scanf("%ld", &a);
if (result == 1) {
  printf("%ld", a);
} else {
  scanf("*[^\n]");
}
```

As a side note, the `%[]` modifier is useful to read in strings containing a certain range of characters only. E.g., you can read `%[a-zA-Z0-9]` to match any sequence of alphanumeric characters.

## Invalid Pointers

Since `scanf` expects the caller to pass in pointers to variables for it to store the results, we need to be careful about what we pass in. It is easy to pass in something like this:

```
long *a;
scanf("%ld", a);
```

The compiler would not warn us since the type matches perfectly. The program may crash since the pointer is not pointing to a valid memory location accessible by the program.

## Buffer Overflow

When we use `scanf` to read a string, it keeps reading until it reaches space, and stores everything that it reads into an array. The problem here is that we do not know when it will stop reading, and therefore how big is the array that we need to allocate for the input!

Let's say we do:

```c
char name[10];
printf("What's your name?", name);
scanf("%s", name);
printf("Hello %s!\n", name);
```

The program would crash if we enter a very long string in the standard input.

You can read a beginners' guide away from scanf() for more information.

## fgets

`fgets` is a better alternative to `scanf` for reading inputs (Note: this is what CS1010 library use internally). `fgets` takes three parameters, a pointer to a string (or buffer), the size of the buffer, and the input to read from (which can be a file, a network socket, or in our case, most of the time `stdin`).

The advantage of `fgets` is that it never overflows the buffer (it knows the size). Once we read the input, we can use functions such as `strtol` or `strtod` to convert the strings to `long` or `double`.

### Avoid atol or atof

Instead of `strtol` or `strtof`, some old school textbooks might show you that you can convert a string to a `long` or a `double` using `atol` or `atof`. You should avoid these two functions (even the man pages of `atof` says so!). They do not provide any mechanism for error checking if the string is not a valid integer or if the input is out of range of the type.

You can read the source code for the CS1010 library to see how it uses `fgets`, `strtol` and related functions to parse numbers and strings from the standard inputs.