【C】Day5

Course	Advanced C
Study Date	@April 11, 2022

【Ch7】Input and Output

7.3 Variable-length Argument Lists

This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way.

Our `minprintf` is declared as

```
void minprintf(char *fmt, ...)
```

The tricky bit is how `minprintf` walks along the argument list when the list doesn't even have a name.

The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is called `ap`, for "argument pointer."

The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

Each call of `va_arg` returns one argument and steps `ap` to the next; `va_arg` uses a type name to determine what type to return and how big a step to take.

Finally, `va_end` does whatever cleanup is necessary. It must be called before the function returns.

7.4 Formatted Input-Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

`scanf` reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments.

The format argument is described below; the other arguments, each of which must be a pointer, indicate where the corresponding converted input should be stored.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found.

On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string.

The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the string according to the format in `format`, and stores the resulting values through arg1, arg2, etc.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters(not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width,

an optional h, l, or L indicating the width of the target, and a conversion character.

As a first example, suppose we want to make a simple calculator

```
#include <stdio.h>

int main() {
  double val, sum;

  while(scanf("%lf", &val) == 1)
    printf("\t%.2f\n", sum += val);

  return 0;
}
```

Suppose we want to read input lines that contain dates of the forms:

```
25 Dec 1988
25/12/1988
```

We can write our code as following:

```
#include <stdio.h>

int main() {
  char *line;
  int month, day, year;
  char monthName[20];
  int length = 20;

  while(getline(&line, &length, stdin)) {
    if(sscanf(line, "%d %s %d", &day, monthName, &year) == 3)
      printf("%s\n", line);
    else if(sscanf(line, "%d/%d/%d", &day, &month, &year) == 3)
      printf("%s\n", line);
    else
      printf("Invalid Input\n");
  }
  return 0;
}
```