SCANF & PRINTF

COMP 2001/2401 course notes

Stephanie Siu & Michael Suen

`printf` and `scanf` are two standard C programming language functions for input and output. Both are functions in the stdio library which means `#include <stdio.h>` is required at the top of your file.

scanf

Function arguments:

```
int scanf(const char * format, [var1], [var2],…)
```

`scanf` requires format placeholders within the format string to denote what data type we would like the input to be read as by the computer.

Here are a table of some of the most commonly used placeholders:

PLACEHOLDER	DATATYPE
%d or %i	integer
%u	unsigned int
%f	floating-point
%lf	double floating-point
%c	character
%s	String

Example 1:

```
int number;
scanf("%d", &number);
```

In this example we have declared an integer called number.
The `"%d"` in `scanf` allows the function to recognise user input as being of an integer data type, which matches the data type of our variable `number`. The ampersand (`&`) allows us to pass the address of variable `number` which is the place in memory where we store the information that scanf read.

Example 2:

```
char word[256];
scanf("%s", word);
```

In this example we have declared a string called word.
The `"%s"` in `scanf` allows the function to recognise user input as being of string data type, which matches the data type of our variable `word`.

Since we're using a string here we require the header file <string.h>, but unlike the example of the integer before, we do not require the ampersand (&) since string is an **array** of characters. An array is already a pointer to a character data type in memory as we have already learned before. Using the ampersand, the above example would be equivalent to:

```
for(int i=0;i<256;i+=1){
    scanf("%s", &word[i]);
}
```

Some extra uses of scanf

In a past assignment we were required to read strings one at a time without the **\n** so that we could print them all on one line.
Square brackets in scanf allow us to define which characters to read from input.

Example 1:

```
scanf("%[JASON]s", string);
```

In this example the program uses scanf to ask user for a string. [JASON] determines the characters the program will only take the characters J, A, S, O and N.
So if the input were "SON", variable S would become "SON".
Whereas if the input were "SANE", variable S would become "SAN".

Example 2:

```
scanf("%[^\n]s", string);
```

This example demonstrates part of the aforementioned assignment. Here we have [^\n]. The not operator (^) is used on the character \n, causes scanf to read everything but the character \n – which is automatically added when you press return after entering input.

scanf also allows us to format our input in various ways. We can ignore preceding white spaces or determine the number of integers we would like to read from the user's input.

Example 3:

```
scanf(" %s", string);
```

In this example white space preceding `%s`, means that any leading whitespaces before the string will be ignored.
So if the input was " whoops", our variable `string` would only take "whoops".

Example 4:

```
scanf("%2d", input);
```

In this example we have a number located between the `%` and `d`, which in this case is 2. The number determines the number of integers our variable `input` (of integer type) will read.
So if the input was "3222", our variable would only read "32".

printf

Function arguments:

```
int printf(const char * format, …)
```

`printf` like `scanf`, also requires format placeholders.

Here are a table of some of the most commonly used placeholders:

PLACEHOLDER	DATATYPE
%d	integer
%u	unsigned int
%f	floating-point
%lf	double floating-point
%c	character
%s	String

Example 1:

```
printf("hello");
```

In this example, the only argument we need in here is the string we're trying to print. This will print to stdout the string input.

Example 2:

```
int number = 9;
printf("%d", number);
```

In this example we have declared an integer called `number`, which takes the value 9. The `"%d"` in `printf` allows the function to recognise the variable `number` as being of integer data type.
Our output would simply be 9.

Example 3:

```
char letter = 'a';
int number = 9;
printf("%c\n%d", letter, number);
```

In this example we demonstrate how to print multiple variables.
Here, `%c` corresponds to our character variable called `letter`, and `%d` corresponds to our integer variable called `number`. We have included `\n` - a newline character – which will print a newline to separate our two variables.
Our output will look like this: `a`
`9`

Example 4:

```
int x = 10;
printf("%5d",x);
```

In this example we have declared an integer `x` with a value of 10. The number located between `%` and `d` represents the indent value. The example above will print : `'    10'`.

Example 5:

```
int* pointer = 5;
printf("%p",pointer);
```

In this example we have declared a pointer to an integer in memory and assigned a value of 5 to it. We use `"%p"` to print to stdout the memory address where pointer points to. This is also the same address that the integer 5 is stored at. Memory addresses that print to stdout are actually

just hex numbers. What will print out is different for every computer and different every time that this function is used.

Special & Important printfs

sprintf

Example 1:

```
int n = sprintf(storageString,"string");
printf("%s",storage);
```

This function is used along with strings. There are at least 2 parameters. The first parameter is where the string will be stored; the second parameter is the formatted string that needs to be printed. The formatted string that is defined in the same way that printf is with its respective placeholders. The possible multi-parameters that are needed in the function are defined the same way as printf. On success, this function returns an integer indicating how long the formatted string is. It is also followed by using printf to print the string to stdout.

fprintf

Example 1:

```
int n = fprintf(storageFile,"string");
```

This function is used along with file reading and writing. Similar to sprintf, it takes in at least 2 parameters and returns an integer indicating how long the formatted string is. One of the big differences between sprintf and fprintf is that the first parameter is not a string where the formatted string will be stored, but a file where the formatted string will be printed. Instead of printing to stdout, the formatted string will appear in the file, similar to the `>out.txt` option in shell command line.