CS102: Standard I/O

Our next topic is **standard input** and **standard output** in C. The adjective "standard" when applied to "input" or "output" could be interpreted to mean "default". Typically, standard output is usually displayed to the monitor, and standard input usually comes from the keyboard.

We have already seen several examples of one of the main standard output functions in C, namely "printf". Remember that "printf" is not really a command; it is a function that is provided for us in the standard input and output library.

The general format of a call to the "printf" function is:

```
printf(format string, data list);
```

The format string is enclosed in quotation marks, and it contains constant text and/or field specifications. A **field specification** describes data that is to be printed. Each field specification in the format string corresponds to a value in the data list. Values in the data list can be constant expressions or they can depend on variables. The first field specification in the format string matches the first value in the data list, the second matches the second, etc.

Each field specification is coded as follows:

```
%<flag(s)><width><precision><size>conversion-code
```

The percent sign and **conversion code** are required but the other four **modifiers** are optional.

There are actually several possible conversion codes, but for now, we will only be concerned with the three we have already seen examples of in class. They are:

c	character
d	integer
f	floating point

The three possibilities for the **size** option are h, l, and L, which stand for short integer, long integer, and long float (double). Examples of the use of size:

```
%hd    /* short integer */
%ld    /* long integer */
%Lf    /* long double */
```

The **precision** option is only used with floats or strings (and we have not covered strings yet). When used to modify a float, the precision tells how many digits should be printed after the decimal point. If the precision option is used, the number of digits must be preceded with a period. Extra digits will be cut, and 0 digits will be added at the right if necessary. If precision is not specified, the default value of 6 is assumed.

The **width** option is used to specify the minimum number of positions that the output will take. If the output would normally take less than the specified number, the output is padded, usually with empty spaces to the left of the value. If the output requires more space than the specified number, it is given the space that it needs. Here are some examples:

```
printf("number=%3d\n", 10);
printf("number=%2d\n", 10);
printf("number=%1d\n", 10);
printf("number=%7.2f\n", 5.4321);
printf("number=%.2f\n", 5.4391);
printf("number=%.9f\n", 5.4321);
printf("number=%f\n", 5.4321);
```

The output of these seven statements in order is:

```
number= 10
number=10
number=10
number=   5.43
number=5.44
number=5.432100000
number=5.432100
```

The first example prints one space to the left of the 10 since a width of 3 was specified. The second case adds no spaces, since the 10 takes up the entire width of 2. In the third case, the specified width is just 1, but the value of 10 requires 2 spaces so it is given them anyway.

In the fourth case, a precision of 2 is specified for a float, so only two digits are printed after the decimal place, and a width of 7 is specified, so the value (which would normally contain 4 characters including the decimal point) has 3 spaces added to the left of the number. In the fifth case, no width is specified, so the value is given exactly what it needs, and a precision of two is specified, but this time, the hundredths digit is rounded up.

In the sixth case, a precision of 9 is specified, so five 0's are added to the end of the value, and in the final case, a default precision of 6 is used, so two 0's are added to the end of the value.

The **flag** option allows one or more print modifications to be specified. Three possible modifications (there are a few others we are not covering) are:

0	fill in extra positions with 0's instead of spaces
-	left justify text instead of right justify text
+	print sign of number (whether positive or negative)

Here are a couple of examples of flags being used:

```
printf("number=%06.1f\n", 5.5);
printf("%-+6.1f=number\n", 5.5);
```

The output of these two statements in order is:

```
number=0005.5
+5.5  =number
```

In the first statement, we are printing a float with precision of 1 and width of 6. Because the 0 flag is used, the three extra positions that need to be filled are filled with 0's instead of spaces. In the second statement, the minus sign causes the value to be left justified (spaces are added to the right instead of the left) and the positive sign causes the sign of the number to be printed with the number.

Field specifications are used to format printed values, often to line things up nicely in columns.

The next statement we will look at is "scanf" which we have also already seen some examples of. The general format for a call to the "scanf" function is:

```
scanf(format string, address list);
```

As with "printf", the format string is enclosed in a set of quotation marks and it contains constant text plus field specifications. With a few exceptions, the format codes used for "scanf" are the same as those used for "printf".

Typically, the format string for a "scanf" will not contain constant text! If it does, that means that the input must contain the exact same text in the same position. For example, consider the following simple program:

```
#include <stdio.h>

int main(void)
{
  int x;

  scanf("Number=%d", &x);
  printf("The value of x is %d.\n", x);

  return 0;
}
```

If the user wants the value of x to be 25, that user would have to type "Number=25" exactly, or the behavior of this little program is unpredictable. To avoid this type of

problem, it is usually not a good idea to include constant text in format strings when using "scanf".

When reading in integers or floats, the "scanf" function skips leading whitespace. That means that all spaces, tabs, and newline characters will be ignored, and "scanf" will keep reading input until it reaches a number. When reading in a character, then "scanf" will read exactly one character, which can be any valid ASCII character or other valid character for the system. If you want to skip a space before a character, then the space has to be explicitly included in the character string. For example, consider the following code (assuming that "a", "b", and "c" are integers and "x" is a character):

```
scanf("%d%d%d%c", &a, &b, &c, &x);
```

Let's say you want "a", "b", "c", and "x" to obtain the values of 1, 2, 3, and 'Z'. Then you would have to type:

```
1 2 3Z
```

If, instead, you type:

```
1 2 3 Z
```

then the value of "x" will be a space!

If you want to be able to enter the line this way, you need to code the "scanf" as follows:

```
scanf("%d%d%d %c", &a, &b, &c, &x);
```

Using spaces between integer field specifications is optional. For example:

```
scanf("%d%d%d", &x, &y, &z);
```

is equivalent to:

```
scanf("%d %d %d", &x, &y, &z);
```

Normally, when reading a numeric value, "scanf" reads until it sees trailing whitespace. If a width modifier is used, it specifies that maximum number of characters to be read. Then "scanf" will read this many characters or until it sees whitespace, whichever happens first.

There are two other reasons that can cause "scanf" to stop.

One is if and **end-of-file character** (**EOF**) is encountered. When reading from an actual file, there is automatically an end-of-file character at the end of the file. When reading from a keyboard, the user can simulate one be pressing a specific character sequence. On Linux machines, for example, the user can enter an EOF by pressing ctrl-d.

The other reason "scanf" may stop is when an invalid character is encountered. For instance, if "scanf" is expecting to read a numeric value and a non-numeric character is encountered, this is an error.

Reasons "scanf" will stop reading a value for a variable:
1) A whitespace character is found after a digit in a numeric sequence.
2) The maximum number of characters have been processed.
3) An end-of-file character is reached.
4) An error is detected.

The "scanf" function returns the number of variables successfully filled in. For example, consider the following program:

```
#include <stdio.h>

int main(void)
{
     int a, b, c;
     int num;

     num = scanf("%d %d %d", &a, &b, &c);
     printf("I have read %d values.\n", num);

     return 0;
}
```

When run, let's say the user types:

```
10 20 30
```

then the program will output:

```
I have read 3 values.
```

If the user types:

```
10 20 hello
```

then the program will output:

```
I have read 2 values.
```

If the user types:

```
hello 10 20 30
```

then the program will output:

```
I have read 0 values.
```

This is not the first time we've seen a function return a value. In a previous lecture, you saw an example where we used "sqrt", for example, which returns the approximate square root of a given number. In fact, we have normally been writing our function "main" to return a 0 to the operating system.

When reading standard input from the keyboard, the input is buffered. In other words, the program in not seeing the text directly as you type it; the characters are being temporarily stored in a buffer somewhere. When you hit "enter", the buffer is sent to the program. Until then, you can edit the buffer by adding (typing) new characters, or by hitting the backspace or delete key to remove the last character from the buffer. These deleted characters will never be seen by your program!

Consider the following simple program:

```
#include <stdio.h>

int main(void)
{
  int x;

  while (1)
  {
    scanf("%d", &x);
    printf("You typed %d.\n", x);
  }

  return 0;
}
```

This program will loop forever, reading numbers from the keyboard and printing them out. Let's say you run this program. You type "10" and hit enter. The program will print "You typed 10." on its own line. Now you type "20" and hit enter. The program will print "You typed 20." Pretty obvious so far.

Now you type "10 20 30" and hit enter. First, the value of 10 is read into "x" and the program prints "You typed 10.". Then the program returns to the "scanf". It does NOT wait for more data to be typed. The remainder of the previous buffer is still accessible to the program! The 20 is read immediately and the program prints "You typed 20.". Then it immediately reads the 30 and prints "You typed 30.".

Now you type "hello" and hit enter. The program tries to read an integer, but it can't, so there is an error. The "scanf" function will be returning 0 here since no variable has been filled in. The value of "x" will remain unchanged, and the program will print "You typed

30." again. Now we get back to the "scanf". The text "hello", from the previous buffer, is still accessible to the computer! Since this is not empty, the program does not wait. It continues to print "You typed 30." over and over again. We are in an annoying infinite loop and must hit Ctrl-C.

If, however, you type "10 20 hello", but without pressing enter, you erase the "hello" with the delete key and type in "30" and hit enter, the program will never see the text "hello" and it will still be in an OK state.

Another thing to note about "scanf" is that the format string should NEVER end with a whitespace character. This will lead to some form of error. For example:

```
scanf("%d\n", &x);
```

This harmless looking code will not work correctly because of the '\n' at the end of the "scanf" format string.

The last thing to remember about "scanf" is that each variable must be preceded by the '&' symbol. The '&' symbol is the **address-of operator**. It takes the address in memory of the variable following the symbol. If we passed the values of the variables to "scanf", "scanf" would be unable to change the values of the variables. By passing the memory address where these values are stored, the function is able to write new values into memory. This should make more sense after covering pointers and functions.

Although it is possible to read and write one character at a time with "scanf" and "printf", two other functions exist specifically for this purpose that are often more readable and friendly functions. These are "getchar" and "putchar", which are also in the stdio library.

Examples of statement calls to these functions are:

```
c = getchar();
putchar(c);
```

The "getchar" function reads one character from standard input and returns its value. The "putchar" function takes one character as a parameter and writes its value to standard output.

So, consider the following program:

```
#include <stdio.h>

int main(void)
{
  char c;

  c = getchar();
  while (c != EOF)
```

```
  {
    putchar(c);
    c = getchar();
  }

  return 0;
}
```

What does this program do? It reads characters from standard input and writes them right back to standard output. Since standard input from the keyboard is buffered, it waits until the user hits enter at the end of a line, and then loops through the characters in the line, printing them out one at a time (including the newline character itself). Since this is extremely fast, the effect that the user sees is that as soon as "enter" is pressed, the line is redisplayed.

This doesn't seem very useful, but in fact this simple program can be useful. So far, we have only looked at cases where standard input comes from the keyboard and standard output goes to the monitor. This can be changed! Normally, in a C program, when we want to read from a file or write to a file, the files are explicitly mentioned within the C code, and different commands that we have not yet learned are used. However, it is possible to declare standard input to come from a file or to have standard output go to a file. This is done from a Linux/Unix/Cygwin command prompt when running the program very simply. Changing where standard input comes from and changing where standard output goes to is often referred to as **redirecting** standard input and standard output.

To redirect standard input to come from a file, you include "<filename" after the call to a program.

To redirect standard output to go to a file, you include ">filename" after a call to a program.

For instance, let's say that last program was called "inout.c" and we compile it creating an executable called "inout". Now try typing:

```
./inout <inout.c
```

We see a copy of the program written to the monitor! The program has just displayed its own source code.

Now try:

```
./inout <inout.c >inout2.c
```

The reads in the lines from "inout.c" and creates a new identical file called "inout2.c". The program has just created a copy of its own source code. (This could also be done with the Linux command "cp".)

The Linux command "diff" shows the difference between two text files. Type in:

```
diff inout.c inout2.c
```

and nothing will be displayed because the two programs are identical!

Note that this type of input and output redirection is NOT part of the C language. The program does not know where standard input is coming from and where standard output is going to. This is controlled by the operating system. The '<' and '>' symbols are telling the operating system to redirect standard input and standard output. It doesn't only work with C programs. For example, if you type:

```
ls >list.txt
```

Then the file called "list.txt" will contain a list of the files in your current directory.

Here is a more interesting example program:

```
#include <stdio.h>

#define IN 1   /* inside a word */
#define OUT 0  /* outside a word */

int main(void)
{
  int nl, nw, nc, state;
  char c;

  state = OUT;
  nl = nw = nc = 0;
  while ((c = getchar()) != EOF)
  {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }

  printf("%d %d %d\n", nl, nw, nc);
}
```

This program counts the number of characters, words, and lines entered from standard input until EOF is encountered. The variable "nc" stores the number of characters. It is incremented every time any character (other than EOF) is read. The variable "nl" stores the number of lines, and it is incremented every time a newline character is encountered. (Therefore, if standard input is coming from a file, and the last line does not have a newline character at the end, this line won't be counted in this count.) Counting the number of words is a little more difficult; if we are in the middle of a word, as long as we don't see whitespace we assume we are still within the word. When we see whitespace, we conclude that we are now outside a word. If we are outside a word and see something other than whitespace, we assume we are now inside a new word and we increase the word count. Since we start with "state" equal to OUT, the first word will get counted. This program is a simplified version of the Unix command "wc"!

Let's say the above program is called "count.c" and the compiled version is called "count". Try running the program on its own source code as follows:

```
./count <count.c
```

The output will be:

```
27 90 453
```

So our program has 27 lines (including blank lines), 90 words, and 453 characters (including newline characters).

If we try running this program on our previous program:

```
count <inout.c
```

the output will be:

```
15 26 157
```

(The exact values might differ depending on how spaces versus tabs are used, where there are blank lines, etc.)

The last thing we will look at now concerning standard input and standard output is the '>>' operator. Like the '>' operator, the '>>' operator redirects standard output to a file, but it first checks to see if the file already exists. If not, it creates the file, and the effect will be the same as using the '>' operator. If so, the new standard output is appended to the end of the preexisting file.