"scanf" - The input function in C!

Just as C provides the output function **printf** in the Standard C Library for processing program output, it also provides the *input function* **scanf** for processing program input.

When you make a call to function **scanf** from within your program, at program execution time the CPU will simply wait at that statement until the user types something in. Once the user types something in, and presses the "Enter" key, then program execution will continue with the next executable statement following the **scanf** statement.

One word of advice here: Always provide the user with some text (prompt message) by using a **printf** statement prior to using your **scanf** statement. If not, the user will have no idea what they are supposed to enter, as the CPU will simply wait there for something to be typed in. So, 99 out of 100 times you will see a **printf** statement before a **scanf** statement in the code.

Let's take a look at an example using **scanf**, then we will explain the function in greater detail.

The following program will prompt the user for 2 numbers, add them together, and output the result.

```
#include <stdio.h>
void main(void)
{
    int value1, value2, sum;
    printf ("Please enter a number: ");
    scanf ("%i", &value1);
    printf ("Please enter a second number: ");
    scanf ("%i", &value2);

    sum = value1 + value2;
    printf ("\nThe sum of %i and %i is %i\n", value1, value2, sum);
    getchar( );
}
```

The output of the above program might look like the following (shown in **blue** print for clarity only). What the user might type in is shown in black.

```
Please enter a number: 100
Please enter a second number: 22
The sum of 100 and 22 is 122
```

Cool huh?!?! :-) (I love this stuff!!!)

In the above sample program execution the user typed in 100 for the first number, then 22 for the second number. The program then continued to calculate the sum and then output the results. As you can see, **scanf** is a very useful function, but not too difficult to use. Notice that I provided a **printf** statement before *each* of the **scanf** statements, otherwise the user would have not idea what to enter.

Before discussing the **scanf** function call, I want to bring up a few *user interface* related issues regarding the "prompt" above. Notice the **printf** statement:

```
printf ("Please enter a number: ");
```

This printf statement did not contain a "**\n**" within its string. Did you notice that? Do you think I accidently forgot it? Never! Each and every character I type in has important meaning! :-) The reason I did not provide a "**\n**" (newline) in the **printf** prompt is because I wanted the user to enter the number *to the immediate right* of the prompt. If I put newline character in the **printf** statement, the prompts to the user would look like:

```
Please enter a number:
100
Please enter a second number:
22
```

Another user-interface issue that I want to point out here is that I put a space between the colon (**:**) and the end quote (**"**) in the prompt. This too was purposely done. If I placed the end quote right up-against the colon, as in the following code:

```
printf ("Please enter a number:");
scanf ("%i", &value1);          ^ notice no space here (not good)
printf ("Please enter a second number:");
scanf ("%i", &value2);                  ^ notice no space here (not
good)
```

The prompts resulting from the above statements of code would look like:

```
Please enter a number:100
Please enter a second number:22
```

Notice no space between the prompt and the data that the user types in. This is not very readable.

Now, we can finally move on to the **scanf** function. (Whew!)

scanf - 2 arguments

As you can see from the above examples, **scanf** requires 2 arguments in its function call. The *first argument i*s the ***format specifier*** (sound familiar?) of the variable where **scanf** will *store* the result typed in by the user; and the *second argument* is the ***name*** of the variable where **scanf** will store the result typed in by the user.

In our example, variable **value1** is of type **int**, so the **%i** format specifier was used in the **scanf** statement. The same is true for variable **value2**.

Note: the special symbol "**&**" is required before the variable name in the second argument in the function call to **scanf**. This is the "**address operator**". It tells **scanf** the "address" of the location of the variable in memory, so that **scanf** knows where to place the data that the user types in. Addresses of variables are also known as "**pointers**". Pointers will be covered during Week 8. All you need to know at this point is that you need to put the address operator before a variable name when calling **scanf**.

I will provide 1 additional example here. The following program prompts the user for an integer (student ID) a floating point number (student grade point average), and a character (middle initial). It then outputs the information back to the user. See how **scanf** processes each one of these.

```
#include <stdio.h>
void main(void)
{
        /* Declare variables */

        int     student_id;
        float   student_gpa;
        char    middle_init, c;

        /* Prompt user for infomation */

        printf ("Please enter your Student ID: ");
        scanf ("%i", &student_id);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */

        printf ("Please enter your Grade Point Average (GPA):
        ");
        scanf ("%f", &student_gpa);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */
```

```
        printf ("Please enter your Middle Initial: ");
        scanf ("%c", &middle_init);
        while ( (c = getchar() != '\n') && c != EOF);   /* something
        new, described below. */

        /* Display info back to user. */

        printf ("\n\nYour Student ID is %i\n", student_id);
        printf ("Your GPA is %.1f\n", student_gpa);
        printf ("Your Middle Initial is %c\n", middle_init);

        getchar( );
    } /* end main */
```

A sample program output of the above code might be:

```
Please enter your Student ID: 200
Please enter your Grade Point Average: 3.5
Please enter your Middle Initial: S


Your Student ID is 200
Your  GPA is 3.5
Your Middle Initial is S
```

First, you notice a new line of code:

```
while ( (c = getchar() != '\n') && c != EOF);
```

This is added to ensure 100% portability. With some systems, the carriage return (\n) is left over in the *input buffer* from the previous scanf (in this case, the GPA). The *input buffer* is a memory location Data is temporarily placed in this input buffer as soon as the user presses the "Enter" key. The **scanf** function then reads the data from the input buffer. With character data, **scanf** will read only a **single character** from the buffer. So, for our example, if you didn't "flush the input buffer", the carriage return left over from the GPA input will remain in the input buffer, and **scanf** will load it (the carriage return) into variable **middle_init**, instead of waiting for you to enter something for the middle initial. (They system thinks you already entered something!)

Also, you want to be sure to get rid of the **\n** that is stored in the input buffer **after** you enter the middle initial. As mentioned, when you scan in a single character, only a single character is read in from the input buffer, leaving your carriage return there. The above line of code "*flushes*" the input buffer (i.e., pulls any remaining characters out of the input buffer), so that there is no data in the input

buffer before, and after the **scanf** for the single character. This ensures 100% portability.

Also notice that there is no "error checking" on user input here. This program would easily "crash" if the user attempted to enter a very large number as the student ID (more than 32,767), or the user tried to enter a number instead of a letter for the middle initial. Normally, we would add code to the program to ensure that the user is entering valid data. This is a very tedious, but necessary process. We will cover some of the basics of "data error checking" throughout this course.

The other point I want to make here is regarding floating point data and **scanf**. Notice that when I prompted for the GPA, I used only "%f" in the call to function **scanf**. I did not specify a number of decimal places in the scanf (e.g., "**%.2f**" or "**%.1f**") ***because we cannot do this in a* scanf**. We cannot control how many decimal places the user will enter during the **scanf** statement. What is *really* important is how many decimal places are displayed during ***output.*** So, we specify the appropriate number of decimal places when we output the information, using the **printf** statement.

If the user entered 3.5 as the GPA value, and the **printf** statement for the GPA was as follows:

```
printf ("Your GPA is %f\n", student_gpa);
```

Then, printf would display all 6 decimal places, since we did not put a "point" and a number between the **%** and the **f** above. So, if the user indeed entered 3.5, the output would be:

```
Your GPA is 3.500000
```

Notice in the full program above that I used "**%.1f**" as the format specifier for the GPA. This ensures that I will see only 1 decimal place in the output even if the user entered more than 1 in the input!

The statement: printf ("Your GPA is %10.2f\n", student_gpa);

Would produce:

```
Your  GPA is      3.50
```

Notice the spacing after the word ‘is’. The 10.2f gives you two decimals in a total field size of 10 spaces. **That type of formatting helps you line up decimal places** in lines of output that may have many different sizes of numbers.

That's about all there is to discuss regarding **scanf** at this time. The next topic discusses another important aspect of C, namely the Standard C Library.