Getting Input From the User by Using *scanf*

Discussion: The most common way to get input from the user is by using the ***scanf*** function.

- But use ***getchar*** or ***getc*** to get input character by character, and use ***fgets*** to get input line by line, and use ***fscanf***, ***fgetc*** or ***fgets*** to get input from a file.

The ***scanf*** function takes as its arguments:

1. A ***format string*** that consists of ***format specifiers*** of the form ***%blah***, separated by spaces, where ***%blah*** is typically one of:
   - `%i` – to input an `int` (i.e., signed integer); equivalently, you can use `%d` for this
   - `%f` – to input a `float` (i.e., single-precision floating-point number)
   - `%lf` – to input a `double` (i.e., double-precision floating-point number)
   - `%s` – to input a `char*` (i.e., string of non-white-space characters)
   - `%u` – to input an `unsigned int` (i.e., an integer whose value is always treated as a non-negative integer)
   - `%li` – to input a `long` (i.e., long integer – but in the environment that we use for CSSE 120, a *long* is no longer than an *int*)
   - `%c` – to input a `char` (but *getchar* is a better choice for character input)

   For example, the format string for inputting an integer, single-precision floating-point number, and a double-precision floating-point number, in that order, would be:

   ```
   "%i %f %lf"
   ```

2. Exactly as many ***addresses*** of variables as there are format specifiers in the format string.
   - Each variable must have type that matches the corresponding format specifier.
   - These arguments must be *addresses* of variables (instead of the variables themselves) so that ***scanf*** can put values into those variables.
     - See Using Pointers to Send Information Back From a Function for further discussion about why the arguments to *scanf* must be addresses of variables instead of the variables themselves.

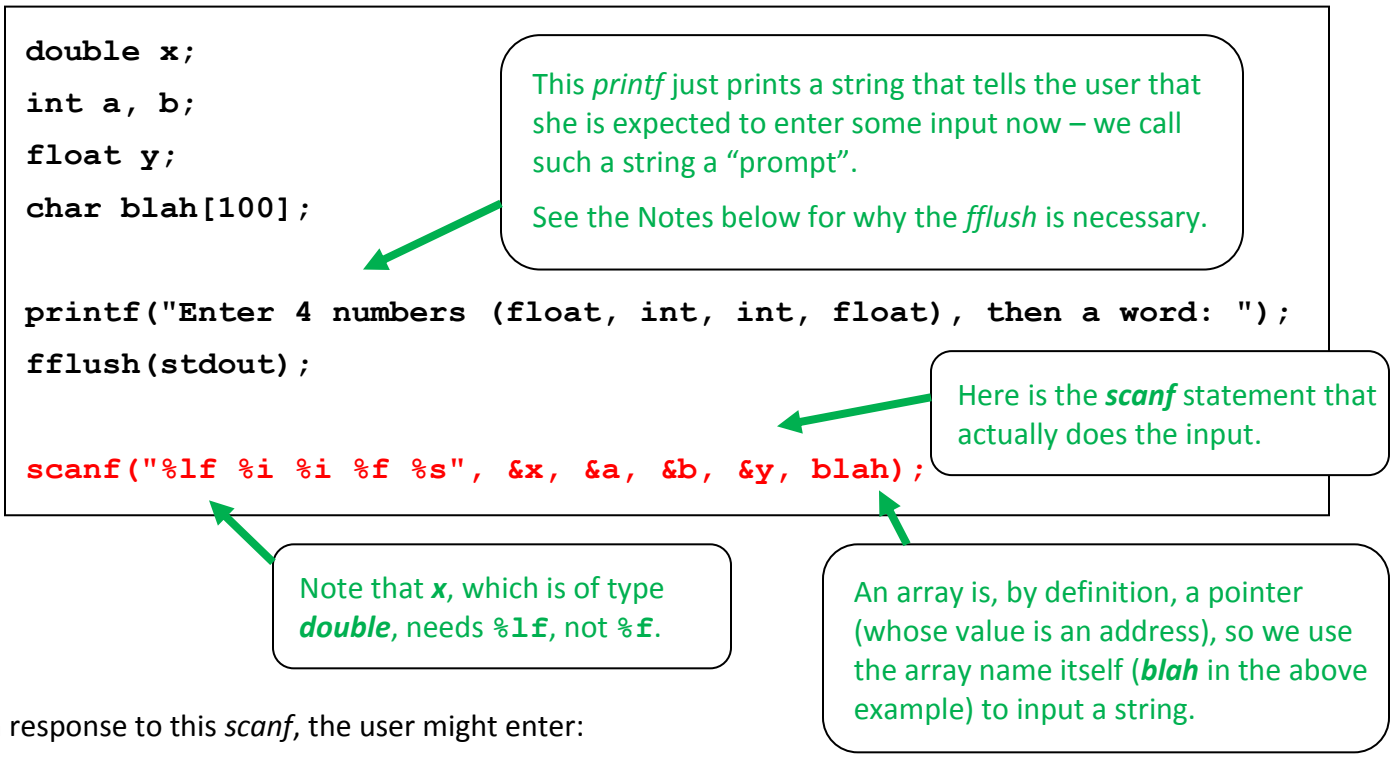
   Continuing the above example in which the user will input an integer, a single-precision floating-point number, and a double-precision floating-point number, in that order, the *scanf* statement would be:

   ```
   scanf("%i %f %lf", &a, &x, &y);
   ```

   where `a` would have previously been declared to be an `int` and `x` and `y` would previously have been declared to be `double`'s.

Getting Input From the User by Using *scanf*

Example: The following code declares variables whose values the user will input, prompts the user to do the input, and then captures the input values in the variables by using *scanf*.



In response to this *scanf*, the user might enter:

```
45.3   -12   8790   4.9999  HelloHowAreYou?
```

Notes:

1. The user can enter the values separated by any white-space, that is, separated by spaces, tabs or newlines, as desired, in any mix. The *scanf* function skips over (and ignores) white-space *except* when doing a `%c` format specifier.
2. If the user enters malformed data (for example, enters "`four`" in response to a `%i`), the program generally continues without an error message but puts garbage data in that variable and in variables subsequently input from the user.
3. Here is why the ***fflush*** is necessary: When you print something using *printf*, the characters to be printed are stored temporarily in a "buffer"; later, they are sent a bunch at a time to the output device (here, the console). This is done for efficiency – you usually don't want to see the characters appear one by one, and it is faster to send them in a burst rather than one by one. A *newline* character normally flushes the buffer. When you want to force a flush, as here, use the *fflush* function.
4. Note that there are 5 format specifiers and 5 addresses of variables that match the format specifiers. Mismatches in number or type will either put garbage values into variables or crash the program when the *scanf* executes.
5. Use **&** to get addresses of non-pointer variables. An array is, by definition, a pointer (whose value is an address), so we use the array name itself (***blah*** in the above example) to input a string. (Equivalently, we could have used `&blah[0]`.)

Gotcha's: The *scanf* function is easily abused, often resulting in hard-to-debug errors:

1. ***Wrong number of arguments:*** If there are not exactly as many addresses of variables as there are format specifiers in the format string, most compilers give no warning. Instead, the program will put garbage values into the variables or crash when the *scanf* executes.

   Example errors:
   ```
   scanf("%i %f", a);
   // Two format specifiers, but only one variable

   scanf("%i", a, b, c);
   // One format specifier, but three variables
   ```

2. ***Wrong type of variable/format specifier:*** If any variable whose address is given does not match its corresponding format specifier, most compilers give no warning. Instead, the program will put garbage values into the variables or crash when the *scanf* executes.

   Example error: using ***%f*** instead of ***%lf*** to input into a variable of type ***double***:
   ```
   double blah;

   scanf("%f", &blah);      // Need %lf here
   ```

3. ***Malformed data:*** If the user enters malformed data (for example, enters "**four**" in response to a **%i**), the program generally continues without an error message but puts garbage data in that variable and in variables subsequently input from the user.

4. ***String overflow:*** When entering a string (format specifier **%s**), if the user enters more characters than the character array has allocated, the extra characters overflow the array without any warning, wiping out whatever happens to be stored in memory after the character array. The result can be an immediate crash, mysterious changes in values of other variables, or a delayed crash – all hard to debug.

   Example error:
   ```
   int age = 12;

   char name[5];

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

   If the user enters a word longer than 4 characters for the `name`, the program may crash or corrupt the `age` variable (or some other variable in the program).

   The input word must be at most 4 (not 5) characters long, to leave space for the `'\n'` that denotes the end of a C string – *scanf* inserts that `'\n'` automatically.

   Note that the program MAY crash or corrupt variables if the input word is too long, but it also might work correctly until changes to the program move the placement of variables in memory. So if your program suddenly stops working correctly, consider the possibility that in code you *previously* wrote, you went past the end of an array.

5. ***Other characters in the format string:*** The first argument to *scanf*, that is, its format string, can be more elaborate than described above. However, until you know more about the format string for *scanf*, stick to the form in the above example: a sequence of ***%blah*** format specifiers, each of which is as described above, separated by spaces.

   Example error:
   ```
   scanf("%i, %f, %i", a, x, b);
         // Avoid commas in the format string - doing so requires
         // that the user put commas in their input.
   ```

Another example: The following code has two functions, each of which gets a floating-point number from the user using *scanf*. Note the two different ways by which the functions send back the input number to the caller.

```
#include <stdio.h>
#include <stdlib.h>

float getInputOneWay();
void getInputAnotherWay(float* b);

int main() {
    float x, y;

    x = getInputOneWay();

    getInputAnotherWay(&y);

    printf("%f %f\n", x, y); // Demonstrates that the function calls worked

    return EXIT_SUCCESS;
}

// Prompt for and input a floating-point number from the user
// and return the number that was inputted.
float getInputOneWay() {
    float a;

    printf("Enter a floating-point number: ");
    fflush(stdout);

    scanf("%f", &a);

    return a;
}


// Prompt for and input a floating-point number from the user and use
// the pointer parameter to send the input number back to the caller.
void getInputAnotherWay(float* b) {
    printf("Enter a floating-point number: ");
    fflush(stdout);

    scanf("%f", b);
}
```

Test your understanding: Do you see why there MUST be an ampersand in the *scanf* in ***getInputOneWay*** but MUST NOT be an ampersand in the *scanf* in ***getInputAnotherWay***?

If not, you might want to reexamine Using Pointers to Send Information Back From a Function.