sscanf

Formatted string input

The formatted string input functions are the opposite of the formatted string output functions. Unlike `printf` and similar functions, which generate formatted output, `scanf` and its friends parse formatted input. Like the opposite functions, each accepts, as a parameter, a template string that contains conversion specifiers. In the case of `scanf` and related functions, however, the conversion specifiers are meant to match patterns in an input string, such as integers, floating point numbers, and character sequences, and store the values read in variables.

- sscanf:
- Formatted input conversion specifiers:

sscanf

The `sscanf` function accepts a string from which to read input, then, in a manner similar to `printf` and related functions, it accepts a template string and a series of related arguments. It tries to match the template string to the string from which it is reading input, using conversion specifier like those of `printf`.

The `sscanf` function is just like the deprecated parent `scanf` function, except that the first argument of `sscanf` specifies a string from which to read, whereas `scanf` can only read from standard input. Reaching the end of the string is treated as an end-of-file condition.

Here is an example of `sscanf` in action:

```
sscanf (input_string, "%as %as %as", &str_arg1, &str_arg2, &str_arg3);
```

If the string `sscanf` is scanning overlaps with any of the arguments, unexpected results will follow, as in the following example. Don't do this!

```
sscanf (input_string, "%as", &input_string);
```

Here is a good code example that parses input from the user with `sscanf`. It prompts the user to enter three integers separated by whitespace, then reads an arbitrarily long line of text from the user with `getline`. It then checks whether exactly three arguments were assigned by `sscanf`. If the line read does not contain the data requested (for example, if it contains a floating-point number or any alphabetic characters), the program prints an error message and prompts the user for three integers again. When the program finally receives exactly the data it was looking for from the user, it prints out a message acknowledging the input, and then prints the three integers.

It is this flexibility of input and great ease of recovery from errors that makes the `getline`/`sscanf` combination so vastly superior to `scanf` alone. Simply put, you should never use `scanf` where you can use this combination instead.

```
#include <stdio.h>

int main()
{
  int nbytes = 100;
  char *my_string;
  int int1, int2, int3;
  int args_assigned;

  args_assigned = 0;

  while (args_assigned != 3)
    {
      puts ("Please enter three integers separated by whitespace.");
      my_string = (char *) malloc (nbytes + 1);
      getline (&my_string, &nbytes, stdin);
      args_assigned = sscanf (my_string, "%d %d %d", &int1, &int2, &int3);
      if (args_assigned != 3)
        puts ("\nInput invalid!");
    }

  printf ("\nThanks!\n%d\n%d\n%d\n", int1, int2, int3);

  return 0;
}
```

Template strings for `sscanf` and related functions are somewhat more free-form than those for `printf`. For example, most conversion specifiers ignore any preceding whitespace. Further, you cannot specify a precision for `sscanf` conversion specifiers, as you can for those of `printf`.

Another important difference between `sscanf` and `printf` is that the arguments to `sscanf` must be pointers; this allows `sscanf` to return values in the variables they point to. If you forget to pass pointers to `sscanf`, you may receive some strange errors, and it is easy to forget to do so; therefore, this is one of the first things you should check if code containing a call to `sscanf` begins to go awry.

A `sscanf` template string can contain any number of any number of whitespace characters, any number of ordinary, non-whitespace characters, and any number of conversion specifiers starting with `%`. A whitespace character in the template string matches zero or more whitespace characters in the input string. Ordinary, non-whitespace characters must correspond exactly in the template string and the input stream; otherwise, a matching error occurs. Thus, the template string `" foo "` matches `"foo"` and `" foo "`, but not `" food "`.

If you create an input conversion specifier with invalid syntax, or if you don't supply enough arguments for all the conversion specifiers in the template string, your code may do unexpected things, so be careful. Extra arguments, however, are simply ignored.

Conversion specifiers start with a percent sign (`%`) and terminate with a character from the following table:

Formatted input conversion specifiers

`c`
: Matches a fixed number of characters. If you specify a maximum field width (see below), that is how many characters will be matched; otherwise, `%c` matches one character. This conversion does not append a null character to the end of the text it reads, as does the `%s` conversion. It also does not skip whitespace characters, but reads precisely the number of characters it was told to, or generates a matching error if it cannot.

`d`
: Matches an optionally signed decimal integer, containing the following sequence:

1. An optional plus or minus sign (`+` or `-`).
2. One or more decimal digits.

Note that `%d` and `%i` are not synonymous for `scanf`, as they are for `printf`.

`e`
: Matches an optionally signed floating-point number, containing the following sequence:

1. An optional plus or minus sign (`+` or `-`).
2. A floating-point number in decimal or hexadecimal format.
    - The decimal format is a sequence of one or more decimal digits, optionally containing a decimal point character (usually `.`), followed by an optional exponent part, consisting of a character `e` or `E`, an optional plus or minus sign, and a sequence of decimal digits.
    - The hexadecimal format is a `0x` or `0X`, followed by a sequence of one or more hexadecimal digits, optionally containing a decimal point character, followed by an optional binary-exponent part, consisting of a character `p` or `P`, an optional plus or minus sign, and a sequence of digits.

`E`
: Same as `e`.

`f`
: Same as `e`.

`g`
: Same as `e`.

`G`
: Same as `e`.

`i`
: Matches an optionally signed integer, containing the following sequence:

1. An optional plus or minus sign (`+` or `-`).
2. A string of characters representing an unsigned integer.
   - If the string begins with `0x` or `0X`, the number is assumed to be in hexadecimal format, and the rest of the string must contain hexadecimal digits.
   - Otherwise, if the string begins with `0`, the number is assumed to be in octal format (base eight), and the rest of the string must contain octal digits.
   - Otherwise, the number is assumed to be in decimal format, and the rest of the string must contain decimal digits.

Note that `%d` and `%i` are not synonymous for `scanf`, as they are for `printf`. You can print integers in this syntax with `printf` by using the `#` flag character with the `%x` or `%d` output conversions. (See printf.)

`s`

Matches a string of non-whitespace characters. It skips initial whitespace, but stops when it meets more whitespace after it has read something. It stores a null character at the end of the text that it reads, to mark the end of the string. (See String overflows with scanf, for a warning about using this conversion.)

`x`

Matches an unsigned integer in hexadecimal format. The string matched must begin with `0x` or `0X`, and the rest of the string must contain hexadecimal digits.

`X`

Same as `x`.

`[`

Matches a string containing an arbitrary set of characters. For example, `%12[0123456789]` means to read a string with a maximum field width of 12, containing characters from the set `0123456789` -- in other words, twelve decimal digits. An embedded `-` character means a range of characters; thus `%12[0-9]` means the same thing as the last example. Preceding the characters in the square brackets with a caret (`^`) means to read a string *not* containing the characters listed. Thus, `%12[^0-9]` means to read a twelve-character string not containing any decimal digit. (See String overflows with scanf, for a warning about using this conversion.)

`%`

Matches a percent sign. Does not correspond to an argument, and does not permit flags, field width, or type modifier to be specified (see below).

In between the percent sign (`%`) and the input conversion character, you can place some combination of the following modifiers, in sequence. (Note that the percent sign conversion (`%%`) doesn't use arguments or modifiers.)

- An optional `*` flag. This flag specifies that a match should be made between the conversion specifier and an item in the input stream, but that the value should *not* then be assigned to an argument.
- An optional `a` flag, valid with string conversions only. This is a GNU extension to `scanf` that requests allocation of a buffer long enough to safely store the string that was read. (See String overflows with scanf, for information on how to use this flag.)

- An optional `'` flag. This flag specifies that the number read will be grouped according to the rules currently specified on your system. For example, in the United States, this usually means that `1,000` will be read as one thousand.
- An optional decimal integer that specifies the maximum field width. The `scanf` function will stop reading characters from the input stream either when this maximum is reached, or when a non-matching character is read, whichever comes first. Discarded initial whitespace does not count toward this width; neither does the null character stored by string input conversions to mark the end of the string.
- An optional type modifier character from the following table. (The default type of the corresponding argument is `int *` for the `%d` and `%i` conversions, `unsigned int *` for `%x` and `%X`, and `float *` for `%e` and its synonyms. You can use these type modifiers to specify otherwise.)

  `h`
  Specifies that the argument to which the value read should be assigned is of type `short int *` or `unsigned short int *`. Valid for the `%d` and `%i` conversions.
  `l`
  For the `%d` and `%i` conversions, specifies that the argument to which the value read should be assigned is of type `long int *` or `unsigned long int *`. For the `%e` conversion and its synonyms, specifies that the argument is of type `double *`.
  `L`
  For the `%d` and `%i` conversions, specifies that the argument to which the value read should be assigned is of type `long long int *` or `unsigned long long int *`. On systems that do not have extra-long integers, this has the same effect as `l`.

  For the `%e` conversion and its synonyms, specifies that the argument is of type `long double *`.

  `ll`
  Same as `L`, for the `%d` and `%i` conversions.
  `q`
  Same as `L`, for the `%d` and `%i` conversions.
  `z`
  Specifies that the argument to which the value read should be assigned is of type `size_t`. (The `size_t` type is used to specify the sizes of blocks of memory, and many functions in this chapter use it.) Valid for the `%d` and `%i` conversions.