A beginners' guide *away from* `scanf()`

Felix Palmen felix@palmen-it.de
2017-06-08

This document is for you if you started to learn programmming in **C**. Chances are you follow a course and the method to read some input you were taught is to use the **`scanf()`** function.

0. What's wrong with `scanf()`?

Nothing. And, chances are, everything for your usecase. This document attempts to make you understand *why*. So here's the very first rule about **`scanf()`**:

> **Rule 0:** Don't use **`scanf()`**. (Unless, you know **exactly** what you do.)

But before presenting some alternatives for common usecases, let's elaborate a bit on the *knowing wha you do* part.

1. I want to read a number from the user

Here is a *classic* example of **`scanf()`** use (and, misuse) in a beginner's program:

```c
#include <stdio.h>

int main(void)
{
    int a;
    printf("enter a number: ");
    scanf("%d", &a);
    printf("You entered %d.\n", a);
}
```

As you probably know, **`%d`** is the conversion for an integer, so this program works as expected:

```
$ ./example1
enter a number: 42
You entered 42.
```

Or does it?

```
$ ./example1
enter a number: abcdefgh
You entered 38.
```

Oops. Where does the value **`38`** come from?

The answer is: This could be any value, or the program could just crash. A crashing program in just two lines of code is *quite easy* to create in **C**. **`scanf()`** is asked to convert a number, and the input doesn't contain any numbers, so **`scanf()`** converts nothing. As a consequence, the variable **`a`** is *never written t* and using the value of an *uninitialized variable* in **C** is *undefined behavior*.

Undefined behavior in C

C is a very low-level language and one consequence of that is the following:

Nothing will ever stop you from doing something completely wrong.

Many languages, especially those for some *managed environment* like **Java** or **C#** actually stop you when you do things that are not allowed, say, access an array element that does not exist. **C** doesn't. As long as your program is *syntactically* correct, the compiler won't complain. If you do something forbidden in your program, **C** just calls the behavior of your program **undefined**. This formally allows anything to happen when running the program. Often, the result will be a crash or just output of "garbage" values, as seen above. But if you're really unlucky, your program will seem to *work just fine* until it gets some slightly different input, and by that time, you will have a really hard time to spot where exactly your program is *undefined*. Therefore **avoid undefined behavior by all means!**.

On a side note, *undefined behavior* can also cause security holes. This has happened *a lot* in practice.

Now that we know the program is broken, let's fix it. Because **scanf()** returns how many items were converted successfully, the next obvious idea is just to retry the "number input" in case the user entere something else:

```c
#include <stdio.h>

int main(void)
{
    int a;
    printf("enter a number: ");
    while (scanf("%d", &a) != 1)
    {
        // input was not a number, ask again:
        printf("enter a number: ");
    }
    printf("You entered %d.\n", a);
}
```

Let's test:

```
$ ./example2
enter a number: abc
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: enter a number: enter a number:
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: enter a number: enter a number:
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: enter a number: enter a number:
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: enter a number: enter a number:
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: enter a number: enter a number:
enter a number: enter a number: enter a number: enter a number: enter a
number: enter a number: enter a number: ^C
```

stooooop! Ok, we managed to interrupt this madness with **Ctrl+C** but *why* did that happen?

Here's a rule:

Rule 1: `scanf()` is not for *reading* input, it's for *parsing* input.

The first argument to **`scanf()`** is a format string, describing what **`scanf()`** should parse. The important thing is: **`scanf()`** never reads anything it cannot parse. In our example, we tell **`scanf()`** to parse a number, using the **`%d`** conversion. Sure enough, **`abc`** is not a number, and *as a consequence, **`abc`** is not even read*. The next call to **`scanf()`** will again find our unread input and, again, can't parse it.

Chances are you find *some* examples saying "let's just flush the input before the next call to **`scanf()`**":

```
fflush(stdin); // <- never do that!
```

Forget about this idea immediately, please.

You'd expect this to clear all unread input, and indeed, some systems will do just that. But *according to **C***, flushing an *input stream* is **undefined behavior**, and this should now ring a bell. And yes, there *are* a lot of systems that won't clear the input when you attempt to flush **`stdin`**.

So, the only way to clear unread input is *by reading it*. Of course, we can make **`scanf()`** read it, using format string that parses any string. Sounds easy.

2. I want to read a string from the user

Let's consider another *classic* example of a beginner's program, trying to read a string from the user:

```
#include <stdio.h>

int main(void)
{
    char name[12];
    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello %s!\n", name);
}
```

As **`%s`** is for strings, this should work with any input:

```
$ ./example3
What's your name? Paul
Hello Paul!

$ ./example3
What's your name? Christopher-Joseph-Montgomery
Segmentation fault

$
```

Well, now we have a *buffer overflow*. You might get **`Segmentation fault`** on a Linux system, any other kind of crash, maybe even a "correctly" working program, because, once again, the program has **undefined behavior**.

The problem here is: **`%s`** matches any string, of any length, and **`scanf()`** has no idea when to stop reading. It reads as long as it can parse the input according to the format string, so it writes a lot more data to our **`name`** variable than the 12 characters we declared for it.

Buffer overflows in `C`

A *buffer overflow* is a specific kind of *undefined behavior* resulting from a program that tries to write more data to an (array) variable than this variable can hold. Although this is *undefined*, in practice it will result in overwriting some *other* data (that happens to be placed after the overflowed buffer in memory) and this can easily crash the program.

One particularly dangerous result of a buffer overflow is overwriting the *return address* of a function. The return address is used when a function exits, to jump back to the calling function. Being able to overwrite this address ultimately means that a person with enough knowledge about the system can cause the running program to execute **any other code** supplied as input. This problem has led to many security vulnerabilities; imagine you can make for example a webserver written in **C** execute your own code by submitting a specially tailored request...

So, here's the next rule:

Rule 2: `scanf()` can be *dangerous* when used carelessly. Always use field widths with conversions that parse to a string (like **`%s`**).

The field width is a number preceeding the conversion specifier. It causes **`scanf()`** to consider a maximum number of characters from the input when parsing for this conversion. Let's demonstrate it in a fixed program:

```
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf("%39s", name);
    printf("Hello %s!\n", name);
}
```

We also increased the buffer size, because there might be really long names.

There's an **important thing to notice**: Although our **`name`** has room for 40 characters, we instruct **`scanf()`** not to read more than 39. This is because a string in **C** always needs a **`0`** byte appended to mark the end. When **`scanf()`** is finished parsing into a string, it appends this byte automatically, and there must be space left for it.

So, this program is now safe from buffer overflows. Let's try something different:

```
$ ./example4
What's your name? Martin Brown
Hello Martin!
```

Well, that's ... outspoken. What happens here? Reading some **`scanf()`** manual, we would find that **`%s`** parses a *word*, not a *string*, for example I found the following wording:

> **`s`**: *Matches a sequence of non-white-space characters*

A *white-space* in **C** is one of *space*, *tab* (**`\t`**) or *newline* (**`\n`**).

Rule 3: Although **`scanf()`** format strings can look quite similar to **`printf()`** format strings, they often have slightly different semantics. (Make sure to **read the fine manual**)

The general problem with parsing "a string" from an input stream is: *Where does this string end?* With **`%s`**, the answer is *at the next white-space*. If you want something different, you can use **`%[`**:

- **`%[a-z]`**: parse as long as the input characters are in the range **`a`** - **`z`**.
- **`%[ny]`**: parse as long as the input characters are **`y`** or **`n`**.

- `%[^.]`: The `^` *negates* the list, so this means parse as long as there is no `.` in the input.

We could change the program, so anything until a *newline* will be parsed into our string:

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf("%39[^\n]", name);
    printf("Hello %s!\n", name);
}
```

It might get a bit frustrating, but this is again a program with possible *undefined behavior*, see what happens when we just press **Enter**:

```
$ ./example5
What's your name?
Hello ÿ¦e!
```

Here's another sentence from a **scanf()** manual, from the section describing the [conversion:

> *The usual skip of leading white space is suppressed.*

With many conversions, **scanf()** automatically skips *whitespace* characters in the input, but with some it doesn't. Here, our *newline* from just pressing enter isn't skipped, and it doesn't match for our conversion that explicitly excludes *newlines*. The result is: **scanf()** doesn't parse anything, our **name** remains *uninitialized*.

One way around this is to *tell* **scanf()** to skip whitespace: If the format string contains any whitespace it matches any number of whitespace characters in the input, including no whitespace at all. Let's use this to skip whitespace the user might enter before entering his name:

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf(" %39[^\n]", name);
    //     ^ note the space here, matching any whitespace
    printf("Hello %s!\n", name);
}
```

Yes, this program works and doesn't have any *undefined behavior**), but I guess you don't like very much that nothing at all happens when you just press enter, because **scanf()** is skipping it and continues to wait for input that can be matched.

*) actually, this isn't even entirely true. This program *still* has *undefined behavior* for empty input. You could force this on a Linux console hitting **Ctrl+D** for example. So, it's again an example for *code you should not write*.

3. Ok, I just want to read *some input* from the user

There are several functions in **C** for *reading* input. Let's have a look at one that's probably most useful to you: **fgets()**.

fgets() does a simple thing, it reads up to a given maximum number of characters, but stops at a newline, which is read as well. In other words: *It reads a line of input*.

This is the function signature:

```
char *fgets(char *str, int n, FILE *stream)
```

There are two very nice things about this function for what we want to do:

- The parameter for the maximum length accounts for the necessary **0** byte, so we can just pass the size of our variable.
- The return value is either a pointer to **str** or **NULL** if, for any reason, nothing was read.

So let's rewrite this program again:

```
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    if (fgets(name, 40, stdin))
    {
        printf("Hello %s!\n", name);
    }
}
```

I assure you this is safe, but it has a little flaw:

```
$ ./example7
What's your name? Bob
Hello Bob
!
```

Of course, this is because **fgets()** also reads the *newline* character itself. But the fix is simple as well: We use **strcspn()** to get the index of the *newline* character if there is one and overwrite it with **0**. **strcspn()** is declared in **string.h**, so we need a new **#include**:

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

int main(void)
{
    char name[40];
    printf("What's your name? ");
    if (fgets(name, 40, stdin))
    {
        name[strcspn(name, "\n")] = 0;
        printf("Hello %s!\n", name);
    }
}
```

Let's test it:

```
$ ./example8
What's your name? Bob Belcher
```

```
Hello Bob Belcher!
```

4. How would I get numbers without `scanf()`?

There are many functions for converting a string to a number in **C**. A function that's used quite often is **`atoi()`**, the name means *anything to integer*. It returns 0 if it can't convert the string. Let's try to rewrite the broken *example 2* using **`fgets()`** and **`atoi()`**. **`atoi()`** is declared in **`stdlib.h`**.

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

int main(void)
{
    int a;
    char buf[1024]; // use 1KiB just to be sure

    do
    {
        printf("enter a number: ");
        if (!fgets(buf, 1024, stdin))
        {
            // reading input failed, give up:
            return 1;
        }

        // have some input, convert it to integer:
        a = atoi(buf);
    } while (a == 0); // repeat until we got a valid number

    printf("You entered %d.\n", a);
}
```

So, trying this out:

```
$ ./example9
enter a number: foo
enter a number: bar
enter a number: 15x
You entered 15.
```

Well, not bad so far. But what if we want to allow an actual **`0`** to be entered? We can't tell whether **`atoi()`** returns **`0`** because it cannot convert anything or because there was an actual **`0`** in the string. Also, ignoring the extra **`x`** when we input **`15x`** may not be what we want.

`atoi()` is good enough in many cases, but if you want better error checking, there's an alternative: **`strtol()`**:

```c
long int strtol(const char *nptr, char **endptr, int base);
```

This looks complicated. But it isn't:

- endptr is set to point at the first character that couldn't be converted. So you have a way to chec whether the whole string was converted.
- base allows you to specify any base you expect a number in. Most of the time, this will be **`10`**, but you could give **`16`** here for parsing hexadecimal or **`2`** for parsing binary.

- **`strtol()`** even sets **`errno`**, so you can check whether a number was too small or too big for conversion.

Now let's use this instead of **`atoi()`** (note it returns a **`long int`**), making use of every possibility to detect errors:

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

int main(void)
{
    long a;
    char buf[1024]; // use 1KiB just to be sure
    int success; // flag for successful conversion

    do
    {
        printf("enter a number: ");
        if (!fgets(buf, 1024, stdin))
        {
            // reading input failed:
            return 1;
        }

        // have some input, convert it to integer:
        char *endptr;

        errno = 0; // reset error number
        a = strtol(buf, &endptr, 10);
        if (errno == ERANGE)
        {
            printf("Sorry, this number is too small or too large.\n");
            success = 0;
        }
        else if (endptr == buf)
        {
            // no character was read
            success = 0;
        }
        else if (*endptr && *endptr != '\n')
        {
            // *endptr is neither end of string nor newline,
            // so we didn't convert the *whole* input
            success = 0;
        }
        else
        {
            success = 1;
        }
    } while (!success); // repeat until we got a valid number

    printf("You entered %ld.\n", a);
}
```

And again, let's try:

```
$ ./example10
enter a number: 565672475687456576574
Sorry, this number is too small or too large.
enter a number: ggggg
enter a number: 15x
enter a number: 0
You entered 0.
```

This looks really good, doesn't it? If you want to know more, I suggest you read on similar functions like **`atof()`**, **`strtoll()`**, **`strtod()`** etc.

5. But *can* I fix the examples with `scanf()` as well?

Yes, you can. Here's a last rule:

> **Rule 4: `scanf()`** is a *very powerful* function. (and with great power comes great responsibility ...)

A lot of parsing work can be done with **`scanf()`** in a very *concise* way, which can be very nice, but it also has many pitfalls and there are tasks (such as reading a line of input) that are much simpler to accomplish with a simpler function. Make sure you **understand the rules** presented here, and if in doubt, read the **`scanf()`** manual **precisely**.

That being said, here's an example how to read a number with retries using **`scanf()`**:

```c
#include <stdio.h>

int main(void)
{
    int a;
    int rc;
    printf("enter a number: ");
    while ((rc = scanf("%d", &a)) == 0)  // Neither success (1) nor EOF
    {
        // clear what is left, the * means only match and discard:
        scanf("%*[^\n]");
        // input was not a number, ask again:
        printf("enter a number: ");
    }
    if (rc == EOF)
    {
        printf("Nothing more to read - and no number found\n");
    }
    else
    {
        printf("You entered %d.\n", a);
    }
}
```

It's not as nice as the version using **`strtol()`** above, because there is no way to tell **`scanf()` not** to ski whitespace for **`%d`** -- so if you just hit **`Enter`**, it will still wait for your input -- but it works and it's a really short program.

For the sake of completeness, if you *really* ***really*** want to get a line of input using **`scanf()`**, of course this can be done safely as well:

```c
#include <stdio.h>

int main(void)
{
    char name[40];
    printf("What's your name? ");
    if (scanf("%39[^\n]%*c", name) == 1) // We expect exactly 1 conversion
    {
        printf("Hello %s!\n", name);
    }
}
```

Note that this final example of course leaves input unread, even from the same line, if there were mor than 39 characters until the newline. If this is a concern, you'd have to find another way -- or just use **`fgets()`**, making the check easier, because it gives you the newline if there was one.