String to Number Conversion in C Takes its Toll

Converting a string to a number in C is no simple affair. Many of you may have heard of atoi(3); one of the ways to convert a string to a number. Unfortunately, modern thinking says that `atoi` should never be used, and so it's use is discouraged. Instead, we now have `strtol`. 1

What's wrong with atoi?

The reason for discouraging use of `atoi` stems from the fact that there is no way to detect if overflow or underflow has occurred, and no way to check if the entire string has been converted (aka there's no way to detect `atoi("123garbage")`). Consider the following code:

```
// 2^32+1, assuming 32-bit int
const char *uintmax_plus_one = "4294967297";
printf("%d\n", atoi(uintmax_plus_one));
```

When run instead of printing `4294967296` as expected, this program will print `1`! The vast majority of programs do not check for or properly handle this case, and so you can end up with situations like the following:

```
// 2^32+1, assuming 32-bit int
const char *uintmax_plus_one = "4294967297";
malloc(atoi(uintmax_plus_one));
```

Now we're allocating far less memory than we expected. This problem can quickly become the source of an integer overflow vulnerability.

Enter strtol!

In order to do the conversion safely, we instead should use `strtol`. It is unfortunately quite difficult to call this function properly. Consider the following documentation pulled from the BSD Library Functions Manual's section on `strtol`:

> The strtol(), strtoll(), strtoimax(), and strtoq() functions return the result of the conversion, unless the value would underflow or overflow. If no conversion could be

> performed, 0 is returned and the global variable errno is set to EINVAL (the last feature is not portable across all platforms). If an overflow or underflow occurs, errno is set to ERANGE and the function return value is clamped according to the following table.

Function	underflow	overflow
strtol()	LONG_MIN	LONG_MAX
strtoll()	LLONG_MIN	LLONG_MAX
strtoimax()	INTMAX_MIN	INTMAX_MAX
strtoq()	LLONG_MIN	LLONG_MAX

Based on this, the two ways to check for an overflow are to check if `strtol` returns 0 or to check if `errno` is set to `ERANGE`. There's another simple case where `strtol` returns 0 specifically if the input string to `strtol` is `"0"`, so in order to accurately detect range errors, we must check for `ERANGE`. This isn't quite so simple either however, as if no error has occurred, `strtol` will not change the value of `errno`. If this happens, and some previous code has set `errno` to `ERANGE` you will erroneously think that a range error has occurred. So now in order to check for range errors you must reset `errno` to a value that indicates that no error has occurred. Now, what value is that? Thankfully, POSIX.1-2008 has considered this possibility, and defined that "No function in this volume of POSIX.1-2008 shall set errno to 0," meaning that no error in all of POSIX will have the value 0. So now we can do the following:

```
errno = 0;
long val = strtol(str, NULL, 10);
if (errno == ERANGE) {
        switch(val) {
        case LONG_MIN:
                // underflow
                break;
        case LONG_MAX:
                // overflow
                break;
        default:
                assert(false); // impossible
        }
} else if (errno != 0) {
        // something else happened. die die die
}
```

But wait, there's more! This works fine for detecting range errors, but fails to detect garbage at the end of the string. Thankfully, `strtol` lets us handle this too via it's `char **endptr` argument.

> If endptr is not NULL, strtol() stores the address of the first invalid character in *endptr. If there were no digits at all, however, strtol() stores the original value of str in *endptr. (Thus, if *str is not `\0' but **endptr is `\0' on return, the entire string was valid.)

Using this, we can detect if there was garbage at the end of the string by passing in a `char **` value for `endptr`.

```
errno = 0;
char *endptr;
long val = strtol(str, &endptr, 10);
if (errno == ERANGE) {
        switch(val) {
        case LONG_MIN:
                // underflow
                break;
        case LONG_MAX:
                // overflow
                break;
        default:
                assert(false); // impossible
        }
} else if (errno != 0) {
        // something else happened. die die die
} else if (*endptr != '\0') {
        // garbage at end of string
}
```

And now we've turned a relatively simple one-line call to `atoi` into twenty lines of code. FML

Enter strtonum

The great folks over at OpenBSD have made a nice replacement for `strtol` which fixes all of the issues discussed. Called strtonum, the function doesn't allow trailing characters and makes it easy to determine if a range error has occurred. The one drawback is that `strtonum` is an OpenBSD extension, and so is not found in any standard. If you want to use `strtonum` on other platforms, you can grab the source here.

1.
`strtol`, short for "string to long" is only one of a few of such functions for converting from a string to a number. Also in this family are strtoimax, strtoll and strtoq. ↩

Recommend Share Sort by Best

Join the discussion…

Droggl • 3 years ago

I wondered for a long time why the heck we need exception handling at all and whats wrong with using the good ole C error handling style. This post illustrates it pretty well ;)

On a more serious note: I always wondered why atoi() was deperecated, thanks for the thorough explanation!

3 • Reply • Share ›

jean luc • a year ago

I wish something like this was posted on reference sites for all functions. It would help more people write safer code.

• Reply • Share ›

William Orr • a year ago

Any thoughts on NetBSD's strtoi(3)?

• Reply • Share ›

Jayson • 3 years ago

Not that it changes the intent of the post, but according to the text you posted from the manual "If no conversion could be performed, 0 is returned AND the global variable errno is set to EINVAL". You should have a check for errno == EINVAL.

• Reply • Share ›

eatnumber1 Mod → Jayson • 3 years ago

That's covered by the errno != 0 portion. I didn't call it out explicity like I did with ERANGE because I didn't explicitly discuss this case like I did with range errors and garbage at the end of the string.

• Reply • Share ›

ikk • 3 years ago

It seems like every line in C turns into 20 lines if you want to do it correctly. Convert a number to string? Check. Allocate or reallocate a block of memory? Check. Make a fool-proof call to snprintf, str(n)cat, strftime, scanf or whatever? Check.

If C functions could do the right thing by default I'd be so happy!

• Reply • Share ›

John → ikk • 3 years ago

http://www.jwz.org/doc/wors...

• Reply • Share ›