C Strings and I/O

The basic nature of string-handling in C causes some problems with input of strings.

The fundamental problems are:

- strings are stored in arrays of char
- these arrays are fixed-length and must be created before the input is read
- input may be unpredictable

Output and C Strings

Assuming a properly-terminated C string, writing it to a file, or standard output, is simple and safe.

The most common approach is to use `fprintf()`:

```
char* str = "some very long string ... ending here";

fprintf(out, "str: %s\n", str);
```

With a properly-terminated string, this operation cannot fail unless the output device is full, which seems unlikely.

Output and C Strings

We can also with `sprint()` and `snprintf()`...

```
char* str = "some very long string ... ending here";

sprintf(str2, "str: %s\n", str);
```

If we've made sure that `*str2` is long enough to hold all the characters we are writing to it, this will be fine.

`snprintf()` allows us to specify a limit on the size of the destination:

```
char* str = "some very long string ... ending here";

snprintf(str2, len2, "str: %s\n", str);
```

fscanf() and Strings

You may use the %s switch in fscanf() to read character data into a char array:

```
#define MAX_LENGTH 25
. . .
char str[MAX_NLENGTH + 1];
. . .
fscanf(in, "%s", str);
```

`fscanf()` will:

- skip leading whitespace,
- read and store characters into `str[]` until whitespace or EOF is encountered,
- write a terminating `'\0'` into `str[]`

BUT, `fscanf()` has no information about the length of `str[]`, so it may write past the end of the array!

This is (arguably) safe when the format of the input date is tightly specified.

fscanf() and Strings

Suppose we want to read personal names from an input file, and we are told each line of the input file will obey the following formatting rule:

```
<first name><\t><middle name><\t><last name><\n>
```

For example:

```
Marion\tMitchell\tMorrison
```

But… how long might one of those strings be?

We have two cases:

a) a maximum length is specified by whatever is supplying the input data

b) in the absence of such guarantee, we can merely make a good guess

fscanf() and Strings

Let's say we decide the maximum name length is 25 characters:

```
#define MAX_NLENGTH 25
. . .
char fname[MAX_NLENGTH + 1];
char mname[MAX_NLENGTH + 1];
char lname[MAX_NLENGTH + 1];

fscanf(in, "%s %s %s", fname, mname, lname);

printf("%s\n%s\n%s\n", fname, mname, lname);
```

```
Marion\tMitchell\tMorrison
```

```
Marion
Mitchell
Morrison
```

OK, that worked as desired…

fscanf() and Strings

Now suppose the input file also contains a city name and a country name, so we have records that are formatted like so:

```
<first name><\t><middle name><\t><last name><\n>
<city name><\n>
<country name><\n>
```

For example:

```
Marion\tMitchell\tMorrison
Winterset
Iowa
```

Now… how long might a city or country name be?

fscanf() and Strings

Let's say we assume our earlier guess is still safe:

```
#define MAX_NLENGTH 25
. . .
char fname[MAX_NLENGTH + 1];
char mname[MAX_NLENGTH + 1];
char lname[MAX_NLENGTH + 1];

fscanf(in, "%s %s %s", fname, mname, lname);
printf("%s\n%s\n%s\n", fname, mname, lname);

char cityname[MAX_NLENGTH + 1];
fscanf(in, "%s", cityname);
printf("%s\n", cityname);

char countryname[MAX_NLENGTH + 1];
fscanf(in, "%s", countryname);
printf("%s\n", countryname);
```

```
Marion\tMitchell\tMorrison
```

```
Marion
Mitchell
Morrison
```

That looks OK…

fscanf() and Strings

But consider the following input data (yes, that's a real place name):

```
Naomi Ellen Watts
Llanfairpwllgwyngyllgogerycchwyrndrobwlllllantysilioggogogoch
Wales
```

```
#define MAX_NLENGTH 25
. . .
char fname[MAX_NLENGTH + 1];
char mname[MAX_NLENGTH + 1];
char lname[MAX_NLENGTH + 1];

fscanf(in, "%s %s %s", fname, mname, lname);
printf("%s\n%s\n%s\n", fname, mname, lname);

char cityname[MAX_NLENGTH + 1];
fscanf(in, "%s", cityname);
printf("%s\n", cityname);

char countryname[MAX_NLENGTH + 1];
fscanf(in, "%s", countryname);
printf("%s\n", countryname);
```

Now we are in trouble.

`cityname[]` is far too small to hold this.

fscanf() and Strings

However, things appear to still be OK. Here's the output from the given code:

```
Naomi
Ellen
Watts
Llanfairpwllgwyngyllgogerycchwyrndrobwlllllantysilioggogogoch
Wales
```

But… let's add some `printf()` statements to check the strings after everything has been read:

```
Naomi
Ellen
ndrobwlllllantysilioggogogoch
Llanfairpwllgwyngyllgogerycchwyrndrobwlllllantysilioggogogoch
Wales
```

Apparently, reading that long place name has caused the array holding the last name to be corrupted… with the tail end of the long place name… and there's no runtime error… just incorrect results…

fscanf() and Strings

So, using `fscanf()` to read character data can lead to silent errors.

It can also lead to runtime errors.

If we merely change the placement of the array declarations in the code shown earlier, execution leads to a segfault…

```
#define MAX_NLENGTH 25
. . .
char cityname[MAX_NLENGTH + 1];
char countryname[MAX_NLENGTH + 1];
char fname[MAX_NLENGTH + 1];
char mname[MAX_NLENGTH + 1];
char lname[MAX_NLENGTH + 1];
. . .
```

Using `fscanf()` to read character data is clearly risky, but can be considered safe if precise assumptions about the input data can be justified.

Reading Delimited Data

Suppose we have an input file with information about music tracks:

```
Buddy Guy   Skin Deep   00:04:30
Eric Clapton   I'm Tore Down   00:03:03
B. B. King  A World Full of Strangers  00:04:22
Eagles   Long Road out of Eden   00:10:17
```

Each line follows the pattern:

```
<artist><\t><track name><\t><track length><\n>
```

Where:

artist	alphanumeric plus spaces, no length limit
track name	alphanumeric plus spaces, no length limit
track length	hh:mm:ss, where h, m and s are digits

Now, `fscanf()` evidently won't do for the artist and track name fields, since they may contain spaces.

fgets(), strtok(), String Library Functions

Here, the strings are delimited by tab characters; can we take advantage of that?

```
Buddy Guy	Skin Deep	4:30
Eric Clapton	I'm Tore Down	3:03
B. B. King	A World Full of Strangers	4:22
Eagles	Long Road out of Eden	10:17
```

`fgets()` can be used to safely read entire lines of character data, if we have a reasonable idea of the maximum length of the line.

`strtok()` can be used break up a character string into chunks, based on the occurrence of delimiting characters.

`strlen()` and `strncpy()` can be used to safely copy the chunks into individual arrays.

`calloc()` and `strlen()` can be used create custom-sized arrays to hold the chunks.

fgets()

```
char* fgets(char* s, int n, FILE* stream);
```

reads bytes from the stream into the array `s` until *n - 1* bytes have been read, or a newline character has been read (and transferred to `s`), or an EOF is encountered.

`s` is then terminated with a zero byte.

returns `s` on success; returns `NULL` if an error occurs or no data is read.

For the input shown below, this code would read the lines sequentially into the array:

```
#define MAX_LINELENGTH 10000     // absurdly large guess
char data[MAX_LINELENGTH + 1];

while ( fgets(data, MAX_LINELENGTH + 1, in) != NULL ) {
   // process the data
}
```

```
Buddy Guy   Skin Deep   4:30
Eric Clapton   I'm Tore Down  3:03
B. B. King  A World Full of Strangers  4:22
Eagles   Long Road out of Eden   10:17
```

strtok()

```
char* strtok(char* s, const char* delimiters);
```

if `s` is not `NULL`:

searches `s` for first character that is not in delimiters; returns `NULL` if this fails.

otherwise notes the beginning of a token, searches `s` for next character that is in delimiters, replaces that with a terminator, returns pointer to beginning of token

if `s` is `NULL`:

performs actions above, using last string `s` passed in, beginning immediately after the end of the previous token that was found

strtok()

Suppose the first line of input shown below has been read into an array `data`:

```
Buddy Guy	Skin Deep	4:30
```

The array contents would be:

'B'	'u'	'd'	'd'	'y'	' '	'G'	'u'	'y'	'\t'	'S'	...

We can use `strtok()` to isolate the artist name, since it's followed by a tab character:

```
char* token = strtok(data, "\t");
```

After the call to `strtok()`, `data[]` looks like this:

'B'	'u'	'd'	'd'	'y'	' '	'G'	'u'	'y'	'\0'	'S'	...

And, `token` points to the first character in `data[]`. . .

. . . and so `token` points to a valid C-string with a terminator.

strtok()

Now, data[] looks like this:

'B'	'u'	'd'	'd'	'y'	' '	'G'	'u'	'y'	'\0'	'S'	...

We can use strtok() again to isolate the title, since it's followed by a tab character:

```
char* token = strtok(NULL, "\t");
```

Now, data[] looks like this:

'\0'	'S'	'k'	'i'	'n'	' '	'D'	'e'	'e'	'p'	'\0'	...

And, token points to the first character in the second token in data[]...

Copying the Token

So, we can identify the artist name, and then copy it into an appropriate array:

```
char* token = strtok(data, "\t");

uint32_t tokenLength = strlen(token);   // get token length

// allocate an array of exactly the right length
char* artist = calloc(tokenLength + 1, sizeof(char));

// copy the token into the new array
strncpy(artist, token, tokenLength);
```

A few points:

- calling `strlen()` is safe because we know the token is terminated
- `calloc()` fills the new array with zeros, so we have a terminator for the new string
- `strncpy()` is safe because the array we are copying into is known to be large enough

Reading the Following Data

Each input line has a length field (time) after the title field.

This is numeric data, and should be read as such.

The interesting part is how to get a pointer to the beginning of the length field:

```
char* lengthField = token + strlen(token) + 1;
```

`strlen(token)` gives us the number of characters in the title field.

We need to add 1 to that to account for the `'\0'` that `strtok()` inserted in place of the tab.

Reading the Following Data

Reading the length data is fairly trivial:

```
int minutes, seconds;

sscanf(lengthField, "%d%*c%d", &minutes, &seconds);
```

The `%*c` specifier accounts for the `':'` that follows the minutes value in the input data.

The single character is read, but discarded.

Putting it all together…

```
char data[MAX_LINELENGTH + 1];
FILE* in = fopen(argv[1], "r");

while ( fgets(data, MAX_LINELENGTH + 1, in) != NULL) {

   char* token = strtok(data, "\t");
   uint32_t tokenLength = strlen(token);
   char* artist = calloc(tokenLength + 1, sizeof(char));
   strncpy(artist, token, tokenLength);

   token = strtok(NULL, "\t");
   tokenLength = strlen(token);
   char* title = calloc(tokenLength + 1, sizeof(char));
   strncpy(title, token, tokenLength);

   char* lengthField = token + strlen(token) + 1;
   int minutes, seconds;
   sscanf(lengthField, "%d%*c%d", &minutes, &seconds);

   printf("Artist: %s\n", artist);
   printf("Title:  %s\n", title);
   printf("Length: %dm %ds\n", minutes, seconds);
   printf("\n");
}

fclose(in);
```

scanset* Format Specifiers

It is also possible to specify a set of characters so that a scan operation is limited to consuming only input characters that occur in that set:

```
char dest[100] = {'\0'};
scanf("%[0123456789]", dest);      // input is "540-231-5605"
```

This would put the characters "540" into the array dest[], properly terminated.

You can also specify the complement of the set by putting '^' at the beginning of the scanset specifier:

```
char dest[100];
scanf("%[^-]", dest);      // input is "540-231-5605"
```

This would also put the characters "540" into the array dest[], properly terminated.

gcc scanset Format Specifiers

gcc also supports using character ranges when specifying a scanset:

```
char dest[100] = {'\0'};
scanf("%[0-9]", dest);       // input is "540-231-5605"
```

This would put the characters `"540"` into the array `dest[]`, properly terminated.

Note that the C Standard does not require this to be supported.

scanset Format Specifiers

Here's a fancier example that processes the entire phone number:

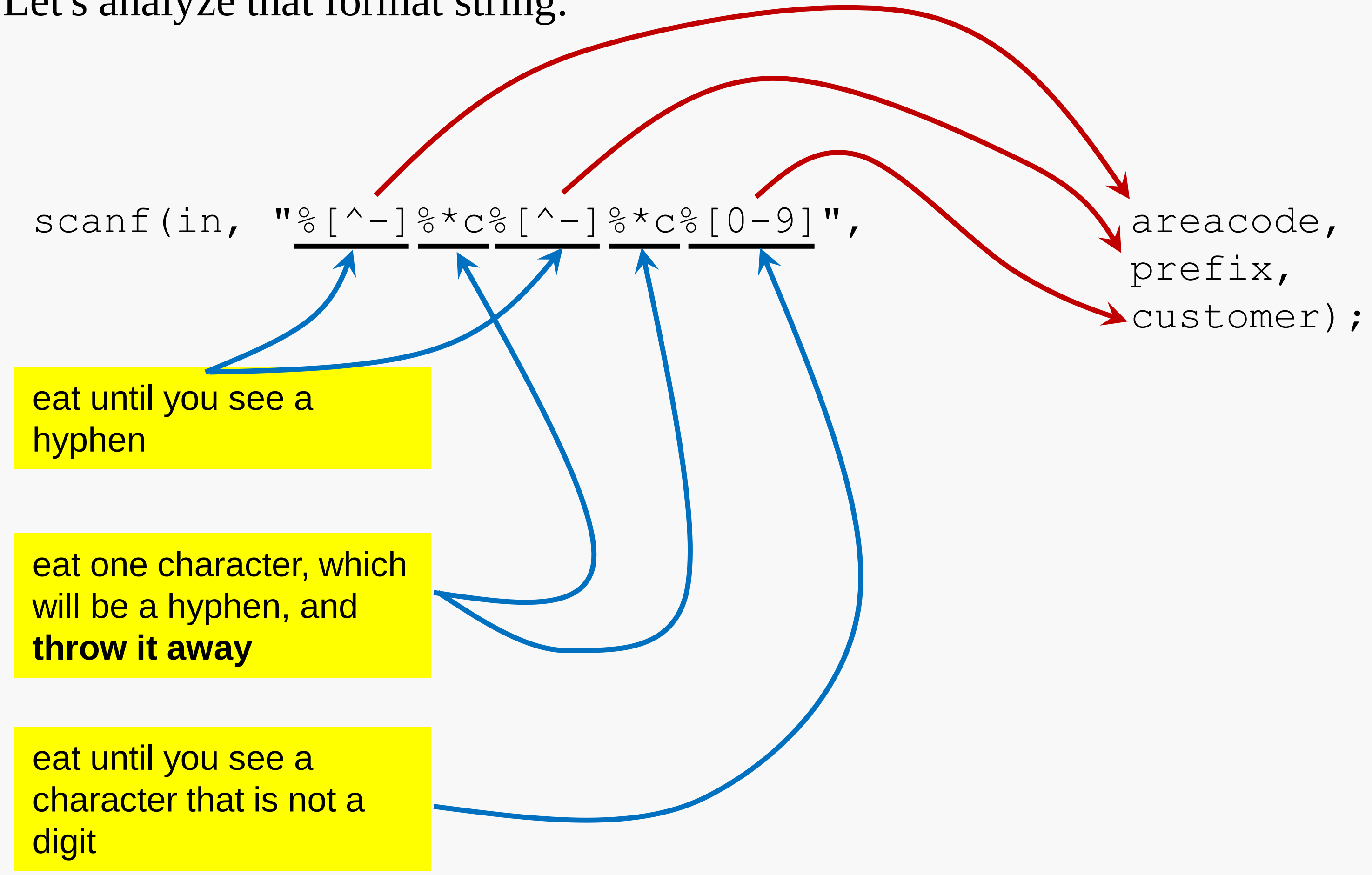
```
char areacode[4] = {'\0'};
char prefix[4]   = {'\0'};
char customer[5] = {'\0'};
                                   // input is "540-231-5605"
scanf(in, "%[^-]%*c%[^-]%*c%[0-9]", areacode, prefix, customer);
```

This would:

- put the characters "`540`" into the array `areacode[]`, properly terminated
- put the characters "`231`" into the array `areacode[]`, properly terminated
- put the characters "`5605`" into the array `areacode[]`, properly terminated

scanset Format Specifiers

Let's analyze that format string:



```
scanf(in, "%[^-]%*c%[^-]%*c%[0-9]",                    areacode,
                                                        prefix,
                                                        customer);
```

eat until you see a hyphen

eat one character, which will be a hyphen, and **throw it away**

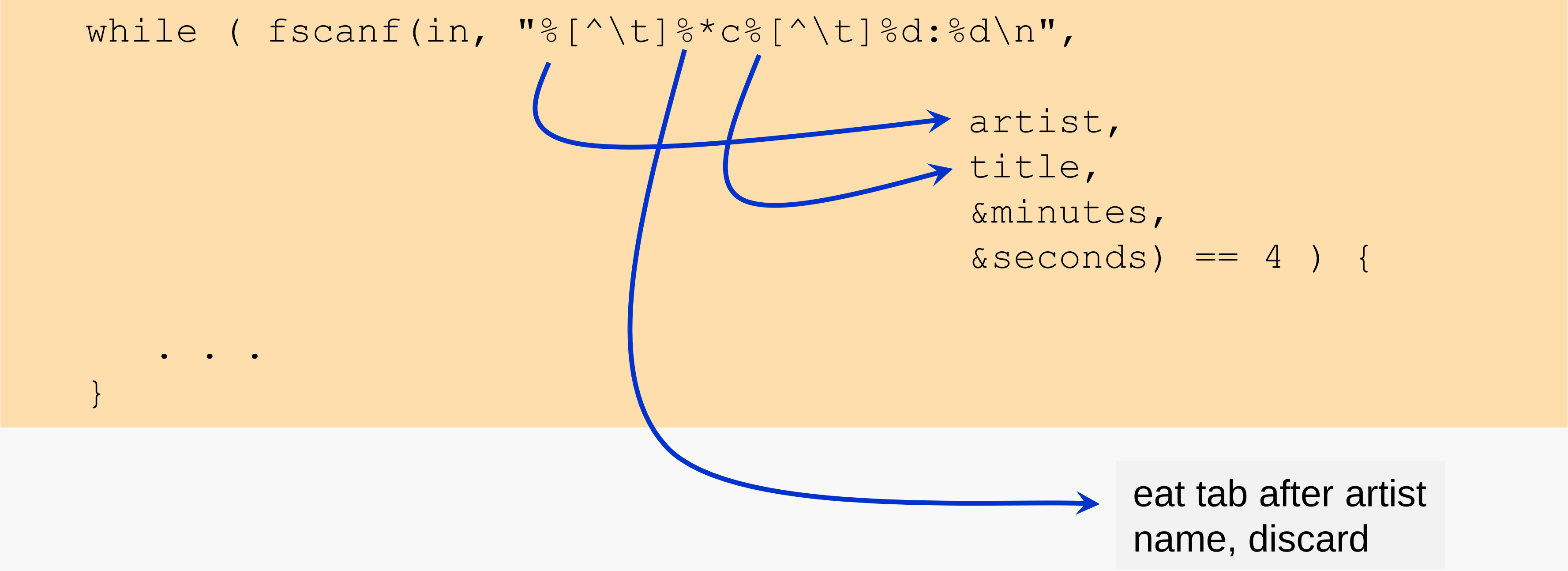
eat until you see a character that is not a digit

Revisiting Music Data with scansets

```
#define MAX_ARTISTLENGTH  100
#define MAX_TITLELENGTH   100

int main(int argc, char** argv) {

   char artist[MAX_ARTISTLENGTH + 1];
   char title[MAX_TITLELENGTH + 1];
   int  minutes, seconds;

   FILE* in = fopen(argv[1], "r");

   while ( fscanf(in, "%[^\t]%*c%[^\t]%d:%d\n",
                       artist, title, &minutes, &seconds) == 4 ) {

      printf("Artist: %s\n", artist);
      printf("Title:  %s\n", title);
      printf("Length: %dm %ds\n", minutes, seconds);
      printf("\n");
   }

   fclose(in);
```

Revisiting Music Data with scansets

```
Buddy Guy    Skin Deep    4:30
Eric Clapton    I'm Tore Down   3:03
B. B. King  A World Full of Strangers   4:22
Eagles    Long Road out of Eden    10:17
```



```
   while ( fscanf(in, "%[^\t]%*c%[^\t]%d:%d\n",
                                           artist,
                                           title,
                                           &minutes,
                                           &seconds) == 4 ) {

      . . .
   }
```

eat tab after artist name, discard