ON STRINGS AND OTHER ACCIDENTS

Using strings in C is one of the world's leading causes of headaches (second only to using dynamic memory in C, but we'll get back to that). In reality, using string in C is not that hard and it can even be fun. All one has to do is to get into the twisted mindset of the *real* C programmer and, pronto!, the string problems is solved. Well... not quite, but close enough.

A.
With string, you don't have to carry around or compute the length in order to execute something on all characters of the string: the last character is a 0, so you can easily write a *while* loop with a pointer that runs through the whole string and stops when it reaches the end

```
char *buf;

(here you write something in buf)

i=0;
while(buf[i]) {
  (do something with the character)
  i++;
}
```

(remember that at the end of the string, buf[i] will be a zero, and that a zero in C is interpreted as *false*). If you are partial to *for* loops, you can also do it like so:

```
char *buf;

(here you write something in buf)

i=0;
for(i=0; buf[i]; i++) {
  (do something with the character)
}
```

Real C programmers, of course, despise indices: they are good for kids, old ladies, and programmers who work in banks using Cobol or Java. The real C programmer likes to do it with pointers

```
char *buf, *pt;

(here you write something in buf)
```

```
i=0;
for(pt=buf; *pt, pt++) {
  (do something with the character *pt)
}
```

*   *   *

B.
Whenever you use the function *strlen* to compute the length of a string, keep in mind that *strlen* does not count the 0 at the end (and correctly so: the length of the string "abc" is three characters, even though in memory it actually occupies four: the three characters that you see, and the 0 that marks the end of the string). If you try to copy a string using the following code, you will get an error, as you will be trying to access an element (the last) that you have not allocated:

```
char *src, *dest;

*dest = (char *) malloc(strlen(buf)); /* AAAAARGHHHH ERROR HERE! */

for (int i=0; buf[i]; i++) {
  dest[i] = buf[i];
} dest[i] = 0; /* This is where you cause the error */
```

This is a rather common error, and can be a rather difficult to detect, because nothing bad will happen when you do *dest[i]=0;*. The error will be revealed possibily many many lines below, for example when you try to release something that you have allocated after allocating *dest*.

By the way... why would you want to go through all that trouble to allocate and copy a string when there is a nice function that does it for you? We'll see it below.

*   *   *

C.
There are two ways of cutting a string while copying it. The first one is simpler but wastes a bit of space (the amount of space that you waste in this kind of things is puny and incinsequential, but it is always a good excuse for the C programmer to write trickier code--never waste a good excuse!). One can copy the whole string and then cut it inserting a 0. For example, if we want to copy the first 10 characters of a string:

```
char *buf = strdup("How I wish I could enumerate pi easily");

char *dest = strdup(buf);
dest[10] = 0; /* Here we cut the string */
```

The part of dest beyond the 10th character and the zero (viz. "I could enumerate pi easily")[1]|is left "dangling", occupying space for nothing. A more elegant solution is to cut *buf* temporarily by inserting a zero in it, copy it, and then fix it:

```
     char *buf = strdup("How I wish I could enumerate pi easily");

     char temp = buf[10];
     buf[10] = 0;
     char *dest = strdup(buf); /* Now it copies only the ten characters that we need */
     buf[10] = temp; /* Fix buf, restoring the character that we had changed. */
```

Note, however, that the latter method can't be used if buf is a constant. It is dangerous, for example, to use it in a function to cut one of the parameters, since the function could be called with the parameter set to a constant.

* * *

D. Real estate people in the US will tell you that if you want to buy a house, the three most important things are: *location, location, and location*. A similar advice holds for strings in C: if you want to do something with string you should first search the standard library for a function that does it, then search the standard library for a function that does it, and finally search the standard library for a function that does it. If you can't find it, implement it.

For example, I am surprised of how many people still do a copy of a string like:

```
     char *dest = (char *) malloc(strlen(src)+1);
     strcpy(dest, src);
```

Now, why would you want to do that when the function

```
     char *dest = strdup(src);
```

does exactly the same thing? Or how many people search the first occurrence of a character in a string as

```
char *search(char *src, char c) {
     char *pt = src;
     while (*pt) {
       if (*pt == c)
         return pt;
            } return NULL; }
```

when you can simply do

[1]The sentence is the beginning of a mnemonic to remember the digits of π: the number of each word represente a digit; a longer version is

> How I wish I could enumerate pi easily, since all these bullshit mnemonics prevent recalling any of pi's sequence more simply

```
     char *pt = strchr(src, c);
```

(there is even a function *strrchr* that finds the *last* occurrence of c, a strstr that looks for a sub-string in a string, and a plethora of other variations.) If you search "C string" in google, you will find a lot of these functions. In Linux, the man pages also have a lot of information. For example, doing "man strchr" you will get a lot of information on strchr and relted functions.

A lot of very good prorgammers worked very hard so that you could have a nicely quirky and very powerful string library: **USE IT!**

*   *   *

Example
In the following, I present you several version of a function that does the same thing: it is given two buffers, *both already allocated*, the first of which is empty while the second contains the name of a file. The function copies the name of the file on the first buffer eliminating the extension.

```
void remove_ext(char *dest, char *buf, int length) {
     int last, i;

     last = -1;
     for (i=0; i<length; i++) {
       if (buf[i]=='.')
         last = i;
     }

     if (last == -1)
       last = length+1;
     for (i=0; i<last; i++) {
       dest[i] = buf[i];
     }
     dest[last] = 0;
```

The function works, but it is hardly something a C programmer would be proud of. The first thing, obvious, is that one doesn't need to pass the length parameter: we can stop all loops when we reach the zero character:

```
void remove_ext(char *dest, char *buf) {
     int last, i;

     last = -1;
     for (i=0; buf[i]; i++) {
       if (buf[i]=='.')
         last = i;
     }
     if (last == -1)
       last = length+1;
     for (i=0; i<last; i++) {
       dest[i] = buf[i];
     }
     dest[last] = 0;
```

Passing *dest* per-allocated and as a parameter is also very very bad: the calling program must allocate this thing without really knowing why. A good C programmer will always let the function that create something take care of allocating it, and will let the funciton return a pointer to whatever it has created:

```
char *remove_ext(char *buf) {
     int last, i;
     char *dest;
     last = -1;
     for (i=0; buf[i]; i++) {
       if (buf[i]=='.')
         last = i;
     }

     if (last == -1)
       last = length+1;
     char *dest = (char *) malloc((last+1)*sizeof(char));
     for (i=0; i<last; i++) {
       dest[i] = buf[i];
     }
     dest[last] = 0;
     return dest;
}
```

Better, but too complicated. The C library has, as we have already seen, a function *strrchr* that searches the last occurrence of a character, and which looks perfect for us. Using this and strdup we can write a reasonably simple copy-and-remove-extension function that allocates a minimum of space:

```
char *remove_ext(char *buf) {
     char *dest, *pt, tmp;
     pt = strchr(buf, '.');
     if (!pt) /* pt is NULL: there is no '.' character */
       return strdup(buf);
     tmp = *pt;
     *pt = 0;
     dest = strdup(buf);
     *pt = tmp;
     return dest;
```

As before, remember that this function will fail if it receives a parameter as a constant. A function that wastes a little memory, but that is very simple and works in any case is the following:

```
char *remove_ext(char *buf) {
     char *dest, *pt;
     dest = strdup(buf);
     pt = strchr(dest, '.');
     if (pt)
       *pt = 0;
     return dest;
```

Easy, *n'est-ce pas*?

Keep in mind that the function is very easy, but the techniques that it exemplifies on the use of pointers will stay with you until the day you retire. And beyond.