

Re: K&R2, exercise 5.4

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2008-04/msg01403.html

- *From:* Philip Potter <pgp@xxxxxxxxxxxxxx>
 - *Date:* Wed, 09 Apr 2008 12:59:32 +0100
-

arnuld wrote:

PURPOSE: see comments or the problem statement from the K&R2
GETTING: 1 as the only output :(

```
/* Exercise 5.4 from K&R2, page 107
 *
 * write the function strend(s, t) which returns 1 if the
 * string t occurs at the end of string s and zero otherwise
 *
 */

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

int strend( char*, char* );

/* main() will only call the function to do the necessary work. */
int main()
```

Please declare main as `int main(void)` or `int main (char argc, char **argv)`. `int main()` is valid but poor style.

```
{
  int arrsize_s, arrsize_t, arrdiff;
```

`arrsize_t` could be a confusing variable name, since so many type names have the `_t` suffix: `size_t`, `ptrdiff_t`, `fpos_t`, etc. It's not wrong, just something to be aware of.

```
char *ps, *pt;
char s[] = "Like a Stone";
char t[] = "Stone";

arrsize_s = (int) sizeof(s) / sizeof(char);
arrsize_t = (int) sizeof(t) / sizeof(char);
arrdiff = arrsize_s - arrsize_t - 1;
```

Re: K&R2, exercise 5.4

This is misleading. If you name a variable `arrdiff`, I expect it to represent a difference. Here, I'd guess the difference between `arrsize_s` and `arrsize_t`. But you instead store `arrsize_s - arrsize_t - 1`. Why?

```
/* we will start at the position in the 1st array, of same length as of
2nd array,
* so that we can compare from there till end. anything before that
length is * of no use to us.
*
*/
ps = s + arrdiff;
pt = t;
```

Now `ps` points to the first character of a string which is one character longer than the one pointed to by `pt`. The `- 1` you added above has introduced an off-by-one error here.

Also, `strend()`'s specification above doesn't say that it expects the strings to be equal length. It should be able to handle arbitrary lengths – even `t` longer than `s`!

```
printf("\n%d\n", strend(ps,pt));
```

```
return EXIT_SUCCESS;
}
```

```
/* I am using pointers because, arrays are never passed to functions and I
* don't want to FAKE the array call
* outer for loop checks for each element of 1st array. * inner for loop
compares elements of both arrays. * if condition checks whether we have
compared the 2nd array till end. */
int strend( char* s, char* t )
{
char *pj;

for( ; *s != '\0'; *s++ )
```

You discard the result of dereferencing `s`. Although `*s++` is not wrong, it's confusing. Just say `s++`. My compiler gave a warning for this.

```
{
printf("s --> %c\n-----\n\n", *s);
for( pj = s; *t == *pj; t++, pj++ )
{
```

```

printf("**t ---> %c\n", *t);
printf("**pj ---> %c\n", *pj);
}

if( *t == '\0' )
{
return 1;
}
}

return 0;
}

```

It seems (I guess) your algorithm is to compare `t` against `&s[0]`, then `&s[1]`, then `&s[2]`, and stop until you get a total match. If you reach the end of `s`, you don't have a match.

There are several problems here:

The reason your `strend()` always returns 0 (or it does on my machine, contrary to your complaint) is that the inner string-comparison loop doesn't terminate when `t` or `pj` reach `'\0'` – so long as they equal each other, the loop continues merrily along until it finds somewhere they /don't/ equal. This means you skip right past the `'\0'` in `t` so by the time you test for it, you've missed it. Even worse, you access beyond the end of the array – which is undefined behaviour. Move the if test into the loop.

You don't reset the value of `t` between outer loop iterations. This means the second time you execute the `pj` loop, `t` no longer points to the first character of the original `t` string, and you've lost the thing you're supposed to be looking for. This means that, for example, if we correct the above problem, you will wrongly match the following:
`s = "sxtring", t = "string"`
because the first 's' will be matched in the first iteration, and the remainder of the string ("tring") will be matched in the third.

Finally, consider your program's efficiency. To compare whether two strings are equal, as you do in the inner 'pj' loop, is an $O(n)$ problem. You do this once for each character in `s`, making the whole algorithm $O(n^2)$ (or it would, if the above problems are all corrected).

There is an $O(n)$ solution to this problem, which involves finding the end of both strings and working backwards. This way you only need one string compare operation.

[Note that the worst case efficiency of $O(n^2)$ requires fairly pathological conditions, for example, `s = "xxxxxxxxxxxxxxxxxxxxx"` and `t = "xxxxxxxxxxxxxxxxxxA"`.]

Philip

.