Lecture 14

CSE 110

24 July 1992

We didn't get a chance to finish working on `type`, but we will soon. In the meantime we talked more about strings and how to work with them.

1 Problem 2.2 (`strrev`) from the Exam

> Write the function `strrev`, whose argument is a string, and which reverses its argument in place. That means that if we do:
>
> ```
> char word[] = "Foo";
> printf("%s\n", word);
> strrev(word);
> printf("%s\n", word);
> ```
>
> the output should be `Foo`, followed by `ooF`.

This was one of the hard problems on the exam. Nobody actually turned in a working solution, although several people came close. First I'll show a working solution, and then we'll discuss some alternative solutions.

One thing a lot of people did was to write a program that actually printed out the string backwards. People who did this correctly got half credit, but it wasn't what the function was supposed to be about—the question asked for a function that would reverse a string 'in place'. That means that we want to be able to tell the function where our string is (presumably via a pointer), and the function will find the data there and reverse it and leave the answer in memory in the same place that the original word was. No input or output was required. This should have been clear from the example: `word` contains `Foo` when we print it out the first time; we run `strrev`, and nothing is printed out until the second `printf`, which demonstrates that `word` now contains `ooF`.

When you're asking yourself how to reverse a string in place, you might think to yourself, "Well, I could copy the string backwards into some auxiliary

space, and then copy the reversed string back from the auxiliary space to the original space." To do that you need to know how to ask for extra space on the fly, and we didn't know how to do that at the time we took the exam.

1.1 A Solution

The solution I was hoping for took some ingenuity to find: We have two pointers, `s` and `e`. `s` starts out pointing to the first character in the string, and `e` starts out pointing to the last character in the string. We swap the characters that `s` and `e` are pointing to; that's easy. Then we increment `s` to point to the second character, and decrement `e` to point to the next-to-last character. We repeat, until both pointers are pointing to the middle letters of the string; then we're done.

Here's the code.

```
void strrev(char *s)
{
  char *e;
  int left=0;                          /* Count of characters left to swap */
  char temp;                           /* For swap */

  /* If length of string is less than 2, don't bother. */
  /* Note short-circuiting here—it's very important. */
  if (s[0] == '\0' || s[1] == '\0')
    return;

  /* First, point e at end of s and compute length of s: */
  for (e=s; *e != '\0'; e++)
    left++;

  /* e now points to NUL character at end of s. */
  /* left is the number of characters we have to swap. */

  e--;
  /* e now points at last character in s. */

  while (left > 1) {
    temp = *s; *s = *e; *e = temp; /* Swap characters at beginning and end */
    s++; e--;                      /* Move towards middle of string */
    left -= 2;                     /* two fewer characters to swap. */
  }
```

```
}
```

There's nothing new here, so it was fair game for the exam.

1.2 A Non-Solution

Some people did think of the auxiliary-space method, and handed in something like this:

```
void strrev(char *s)
{
  int i, len = strlen(s);
  char aux[len+1];              /* You can't do this */

  aux[len] = '\0';

  for (i=0; i<len; i++)
    aux[i] = s[len-1-i];        /* Copy string backwards */

  for (i=0; i<len; i++)
    s[i] = aux[i];              /* Copy string forwards */

  return;
}
```

This is ingenious, but it doesn't work. The catch is that an array's size must be determined at the time the program is compiled.[1] Nevertheless, people who did this correctly (not counting the illegal array declaration itself) got three-quarters credit.

1.3 Another Solution with `strdup`

Here's another auxiliary-space solution—this one does work. Unfortunately, nobody could have turned this in because we hadn't covered `strdup` by the time we had the exam:

[1] Some compilers do allow variable-sized arrays, as a non-portable extension.

```
void strrev(char *s)
{
  int i, len = strlen(s);
  char *aux;

  aux = strdup(s);

  ... /* Same as above */
}
```

`strdup`'s argument is a string. `strdup` does this:

1. It looks at the string,
2. It finds out (probably with `strlen`) how much space is necessary to store the string, (one byte for each character in the string, and an extra byte for the `NUL` character that terminates it),
3. It somehow finds and reserves enough space for a copy of the string,
4. It copies the string into the memory it's reserved, and
5. It returns a pointer to the first character in the new copy of the string.

If `strdup` fails, for example because it can't reserve enough free space for a copy of its argument, it returns the `NULL` pointer.

When we say that `strdup` 'reserves' space, we mean that the space won't be used for something else until we tell the computer that it's all right to do so. We can be sure that future calls to `strdup` will find other space, and that variables we declare will be allocated out of space other than that reserved by `strdup`.

1.4 `free`

The space reserved by `strdup` stays reserved until the program terminates or until we explicitly make the space available for re-use. This is called *freeing* the space; we do it with the `free` function. If `p` is a <pointer to `char`> which points to space reserved by `strdup`, the call `free(p)` frees that space; a variable might get put there later, or a future call to `strdup` might copy new data there and return a new pointer to it.

2 Comparing Characters and Strings

The comparison operators `<`, `<=`, `>`, and `>=` all work on characters as well as on numbers, in a reasonable way.

2.1 Comparing Single Characters

It's definitely true that:

```
'0' < '1' < ... < '8' < '9'
'A' < 'B' < ... < 'Y' < 'Z'
'a' < 'b' < ... < 'y' < 'z'
```

The relative ordering of these three classes, however, is implementation-dependent. Depending on what kind of machine you are using, it might be true either that`'A' < 'a'` or that `'a' < 'A'`. On the machines we use, it's the case that `'9' < 'A' < 'Z' < 'a'`.

The `NUL` character is always less than any printable character. A printable character is one that makes a space or a mark on the screen.

We can use this character ordering to write a function that compares strings alphabetically.

2.2 Comparing Strings

We'll write a function which accepts two strings as arguments and returns some kind of answer to say which string comes earlier in an alphabetical ordering. We'll do something reasonable for nonalphabetic strings.

We'll say that a string is *less* than another string if it comes before that string in the dictionary. For example: `bar` is less than `foo`; `food` is less than `fool`; `fool` is less than `foolish`. Please note that 'less' does not mean 'shorter'. `foolish` is less than `zebra`, but it is not shorter than `zebra`.

Our function will examine the first character of each argument. If the characters are different, we know right away which string is least: The one whose first character is least. We can return an answer right away in this case. We'll return `-1` if the first argument is less than the second, and `1` if the first argument is greater than the second.

If the first two characters are equal, we'll go look at the second two characters, and soforth.

We keep doing this until either we find two characters that don't match (in which case we return `1` or `-1` as above) or until we reach the end of one or both strings.

If we reach the end of both strings at once, then they were identical, and we return `0`.

Otherwise, one of the strings is a *prefix* of the other, which means that it is just the same as the other string, only it is missing some characters off the end; for example, `foo` is a prefix of `foolish`. In this case the shorter string is least, and we should return `1` or `-1` as above.

That said, here's the code we wrote in class:

```
int strcmp(char *s1, char *s2)
{
  while(1) {
    if ( *s1 == '\0' && *s2 == '\0' )
      return 0;                 /* strings are identical */
    else if (*s1 == '\0')
      return -1;                /* s1 is a prefix of s2 */
    else if (*s2 == '\0')
      return 1;                 /* s2 is a prefix of s1 */
    else if ( *s1 < *s2 )
      return -1;                /* s1 is less */
    else if ( *s1 > *s2 )
      return 1;                 /* s2 is less */
    else {
      s1++; s2++;
    }
  }
}
```

This code works. It's functionally equivalent to a standard library function called `strcmp`, which operates in the same way. `strcmp` returns negative, zero, or positive, depending on whether its first argument is less than, equal to, or greater than its second argument. (`strcmp` doesn't always return −1, 0, or 1.) To use `strcmp`, you must include `<string.h>`.

2.3 Case-Insensitive Comparison

Our `strcmp` function does a thing that we might find peculiar: It says that `Snider` is less than `food`, because, in Turbo-C++, capital letters are always less than lowercase letters. We say that `strcmp` is *case-sensitive*, because it treats the words `Snider` and `snider` differently. We might wish to erase this distinction and make `Snider` greater than `food`, because `s`, capital or not, comes after `f` in the dictionary.

To do this, we can use the standard function `tolower`. `tolower` accepts one argument, which is a character. If the character is not an uppercase letter, `tolower` returns the character it was passed; if the argument was an uppercase letter, `tolower` returns the lowercase version of that letter. For example, `tolower('X')` is `'x'`; `tolower('y')` is `'y'`; `tolower('2')` is `'2'`.

What we'll do is convert the characters in our strings to lowercase before we compare them; then the `S` in `Snider` will behave as though it were a lowercase S.

Replace the lines

```
else if ( *s1 < *s2 )
  return -1;
else if ( *s1 > *s2 )
  return 1;
```

with

```
else if ( tolower(*s1) < tolower(*s2) )
  return -1;
else if ( tolower(*s1) > tolower(*s2) )
  return 1;
```

. The function we get is called `strcasecmp` on UNIX systems, but it seems to be called `stricmp` or `strcmpi` under Turbo-C++.

`tolower`, and a collection of similar functions, such as `toupper`, are declared in `<ctype.h>`.