Faculty of Computer Science, Dalhousie University *2-Nov-2018*

CSCI 2132 — Software Development

Lecture 25: String Library Functions

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Finished mergesort2 "fill-in the blanks"
- Strings:
- String literals
- String variables
- Reading and writing strings

Slide notes:

Aside: A Common Mistake

```
/* Example vla.c */
#include <stdio.h>
int main() {
  int n;
  scanf("%d", &n);
  /* double a[n];     /* Wrong! */
  while (n > 0) {
    double a[n];      /* Correct! */
    printf("Size of double: %d\n", sizeof(double));
    printf("Size of array: %d\n", sizeof(a));
    printf("Length of array: %d\n",
                  sizeof(a) / sizeof(double));
    /* Array can be used here */
    scanf("%d", &n);
  }
  return 0;  }
```

If we do not want to ignore white-space characters, we can use the `%c` specifier, as in the following example:

```
char s[20];
scanf("%19c", s);
s[19] = '\0';
```

The above specifiere `%19c` will read exactly 19 next characters, without ignoring white-space characters, and put them into the array `s`, and it will not append the input with the null character. The specifiers which accept only some characters from a set, such as `%[a-z]`, or complement of a set, such as `%[^0-9]`, will fill a string variable with read characters and append a null character at the end. They do not ignore white-space characters. Again, to make sure that they do not overfill the given variable, we can use the maximal field width, as in:

```
char s[20];
scanf("%19[a-zA-Z]", s);
```

Reading a Line

There is another function called `gets` which can be used to read one entire line from the user and store it in a string. However, it does not check whether the string variable used to store this line has enough capacity. Thus, it is unsafe to use it, and if you do use it, gcc will issue a warning. If you need to use a function that reads a line in similar manner, you should use the function `fgets`. The function `fgets` is actually meant to be used with files and it expects a file argument, but since `stdin` is a file handle allocated for reading the standard input we can use `fgets` as in the following example:

```
char line[1000];
fgets(line, 1000, stdin);
```

In the above example, `fgets` will read the standard input until it sees a new-line character, and it will store the complete line, followed by a null character into the variable `line`. If during reading, it reads 999 characters, it will stop reading and append the null character, thus properly filling out the variable `line`.

Slide notes:

Buffer Overflow Risks

- Happens if we read input without length control, for example:
  ```
  char buffer[1000];
  gets(buffer);
  /* or */
  scanf("%s", buffer);
  ```
- Bug, but may also be a risk of security attack
- A malicious user can provide a long, carefully crafted input to overwrite stack and cause program to start executing injected machine code

Example of a User-Implemented Read Function

We often design our own input function to read strings. The advantage is that we can control the exact program behavior, especially what to do when the string to be read is too long to fit in the string variable that is used to store it. In the following example, we will write a function that reads characters without skipping white-space characters until a newline is reached, which is not stored in the string. This function returns the number of characters stored. This function is an example from the textbook (King), and it is very similar the function `fgets` mentioned above. (There is a mistake in the textbook code, which is corrected here. There are a couple of additional improvements.)

```
/* Read one line from stdin and store in string str of
   capacity n.
   The null character is included in the capacity n.
   If the line is too long, only a part is read.
*/
int read_line(char str[], int n) {
   int ch, i = 0;

   while ( (ch = getchar()) != '\n' && ch != EOF )
       if (i < n-1) str[i++] = ch;

   str[i] = '\0';

   return i;
}
```

Here we use n as the maximum number of characters to be stored in `str`, excluding null. If the number of characters in the input line is too large, the program reads and then discards the remaining characters. You may change this behavior depending on your particular application.

20.4 String Library Functions

There are a set of standard library functions for strings. Their prototypes are provided in `string.h`. Thus to use them, we need to include the following include line at the beginning of the program:
`#include <string.h>`

One such function is `strcpy`, which copies string `s2` into `s1` and returns `s1`:

```
char* strcpy(char *s1, const char *s2);
```

Here the keyword `const` means that even though the address of object that `s2` points to is passed to this function, the function `strcpy` does not change this object.

Let us use pointers to implement this function as an exercise:

```
char* strcpy(char *s1, const char *s2) {
  char *p = s1;

  while ((*p++ = *s2++) != '\0')
    ;

  return s1;
}
```

The interesting part is the while loop which has a null statement. It is very concise and is popular among C programers. We could certainly also use a few statements to perform the same task. One homework for you is to implement this function using array subscripting instead of pointer arithmetic.

To use this function, we need make sure that `s1` points to a sufficiently large block of memory. The following example would NOT work:

```
char s1[] = "abc";
strcpy(s1, "defg");
```

To make this example work, we need declare `s1` as a character array of 5 or more elements.

Another function is `strcat`, and it appends string `s2` to `s1` and returns `s1`:

```
char* strcat(char *s1, const char *s2);
```

Again we need to make sure that `s1` points to a sufficiently large block of memory.

The third function is `strcmp`, which compares strings `s1` and `s2` lexicographically.

```
int strcmp(const char *s1, const char *s2);
```

To determine whether `s1` is less than `s2` in lexicographic order, we find the first position at which these two strings differ, and the result of the comparison is the result of comparing the two characters from these two strings stored at this position.

If `s1 < s2`, this function returns a negative value. If `s1 == s2` this function returns 0. Otherwise, a positive value is returned. If you think it terms of numbers, you can thing of this result as returning the difference $s1-s2$—it is negative if `s1 < s2`, zero if `s1 == s2`, and positive if `s1 > s2`.

For example, if we call `strcmp("large", "little")`, then a negative value will be returned. This is because `"large"` is lexicographically smaller than `"little"`. These two words start with the same character, but for the next position, we have `a < i`.

Finally, one common string function is `strlen`, which computes the length of a string:

```
size_t strlen(const char *s1);
```

What is new here is the type `size_t`: This is an implementation-defined unsigned integer type used for the size of objects in memory. It is implementation-defined because its range depends on the platform. We often assign the return value of `strlen` it to an `int` variable, unless we are processing very large strings that require gigabytes to store.