Frequently Used Functions From `string.h`

Pedram Pasandide

Introduction

The `string.h` library in C provides various functions for manipulating strings and memory. Here's a list of frequently used functions available in `string.h`:

String Manipulation

1. `strlen`: Returns the length of a string (excluding the null terminator).

   Function Format:

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

   where `strlen` is the name of the function with input argument `*str` and type `const char`. The `*` before `str` indicates that `str` is a pointer. We will discuss pointers later in this course. Here, we just learn how to use the function. The only thing we need to know at this moment is that `*str` is a string.

   Similar to `int main()` function where `int` is the type of data the function `main` returns, here the returned value from `strlen` has the type `size_t`. What is `size_t`?

Note: `size_t` data type is specifically designed to represent the size of objects in memory or the result of the `sizeof()` operator. Remember the place holder for `sizeof()` is `%zu`. `size_t` is an `unsigned integer` type, guaranteed to be large enough to hold the size of the largest object the platform can support. Always an unsigned integer, meaning it cannot store negative values. This makes it ideal for indices, sizes, or counts (which are inherently non-negative).
`size_t` range depends on the architecture (32-bit or 64-bit):

- On a 32-bit system: Typically 32 bits (range: 0 to 4,294,967,295).
- On a 64-bit system: Typically 64 bits (range: 0 to 18,446,744,073,709,551,615).

`size_t` is guaranteed to represent the size of any object in memory. Take `int` as an example with range typically from -2,147,483,648 to 2,147,483,647. This means `int` cannot safely represent sizes of large objects on 64-bit platforms.
Usage Example:

```
#include <stdio.h>
#include <stddef.h>  // For size_t

int main() {
 size_t size = sizeof(int) * 10;  // Always non-
    negative
 printf("Size of array: %zu bytes\n", size);  // %zu is
     the correct format specifier for size_t
 return 0;
}
```

Example:

```
int main() {
 char str[] = "Hello, McMaster!";
 printf("Length of string: %zu\n", strlen(str));
 return 0;
}
```

2. `strcpy`: Copies a string (including the null terminator) from the source to the destination.

   Function Format:

```
char *strcpy(char *dest, const char *src);
```

`strcpy` has has two input arguments including `char *dest` and `const char *src`. `dest` is a pointer to `char` (a string) shown by `*`. It is not `const` because it will be changed. However, `src` is `const`.

`char *` before `strcpy` indicates the return type of the function, meaning that `strcpy` returns a pointer to a `char`. The function updates `*dest` and returns the pointer `dest`. I don't want to go too deep into this topic since we haven't covered pointers yet. For now, we just want to learn how to use it. You can revisit this section after we've discussed pointers, and it will make more sense to you then. *Example*:

```
int main() {
 char src[] = "Hello, McMaster!";
 char dest[50];
 strcpy(dest, src);
 printf("Copied string: %s\n", dest);
 return 0;
}
```

3. `strncpy`: Copies up to a specified number of characters from the source string to the destination.

   Function Format:

   ```
   char *strncpy(char *dest, const char *src, size_t n);
   ```

   Example:

   ```
   int main() {
    char src[] = "Hello, McMaster!";
    char dest[10];
    strncpy(dest, src, 5);
    dest[5] = '\0';   // Ensure null termination
    printf("Copied string: %s\n", dest);
    return 0;
   }
   ```

4. `strcat`: Concatenates two strings (appends the source string to the destination string).

   Function Format:

   ```
   char *strcat(char *dest, const char *src);
   ```

   Example:

```c
int main() {
 char str1[50] = "Hello";
 char str2[] = ", McMaster!";
 strcat(str1, str2);
 printf("Concatenated string: %s\n", str1);
 return 0;
}
```

5. `strncat`: Concatenates up to a specified number of characters from the source string to the destination.

   Function Format:

```c
char *strncat(char *dest, const char *src, size_t n);
```

   Example:

```c
int main() {
 char str1[50] = "Hello";
 char str2[] = ", McMaster!";
 strncat(str1, str2, 7);  // Append only 7 characters
 printf("Concatenated string: %s\n", str1);
 return 0;
}
```

String Comparison

1. `strcmp`: Compares two strings lexicographically.

   Function Format:

```c
int strcmp(const char *str1, const char *str2);
```

   Example:

```c
int main() {
 char str1[] = "Hello";
 char str2[] = "McMaster";
 int result = strcmp(str1, str2);
 printf("Comparison result: %d\n", result);
```

```
    return 0;
}
```

2. `strncmp`: Compares up to a specified number of characters of two strings lexicographically.

   Function Format:

```
char *strchr(const char *str, int c);
```

   Example:

```
int main() {
 char str1[] = "Hello";
 char str2[] = "Help";
 int result = strncmp(str1, str2, 3);
 printf("Comparison result: %d\n", result);
 return 0;
}
```

String Searching

1. `strchr`: Searches for the first occurrence of a character in a string.

   Function Format:

```
char *strchr(const char *str, int c);
```

   Example:

```
int main() {
 char str[] = "Hello, McMaster!";
 char *pos = strchr(str, 's');
 if (pos) {
  printf("Found character at: %s\n", pos);
 } else {
  printf("Character not found\n");
 }
 return 0;
}
```

2. `strrchr`: Searches for the last occurrence of a character in a string.

   Function Format:

```
char *strrchr(const char *str, int c);
```

Example:

```
int main() {
 char str[] = "Hello, McMaster!";
 char *pos = strrchr(str, 'o');
 if (pos) {
  printf("Last occurrence: %s\n", pos);
 } else {
  printf("Character not found\n");
 }
 return 0;
}
```

Note: The `*` before `pos` indicates that `pos` is a pointer. We will discuss pointers later in this course.

3. `strstr`: Finds the first occurrence of a substring in a string.

   Function Format:

   ```
   char *strstr(const char *haystack, const char *needle);
   ```

   Example:

   ```
   int main() {
    char str[] = "Hello, McMaster!";
    char *pos = strstr(str, "McMaster");
    if (pos) {
     printf("Substring found at: %s\n", pos);
    } else {
     printf("Substring not found\n");
    }
    return 0;
   }
   ```

4. `strtok`: Splits a string into tokens based on a delimiter.

   Function Format:

   ```
   char *strtok(char *str, const char *delim);
   ```

   Example:

```c
int main() {
 char str[] = "Hello, McMaster! How are you?";
 char *token = strtok(str, " ");
 while (token != NULL) {
  printf("%s\n", token);
  token = strtok(NULL, " ");
 }
 return 0;
}
```

String and Memory Handling

1. `memcpy`: Copies a specified number of bytes from one memory location to another.

   Function Format:

   ```c
   void *memcpy(void *dest, const void *src, size_t n);
   ```

   Example:

   ```c
   int main() {
    char src[] = "Hello, McMaster!";
    char dest[50];
    memcpy(dest, src, strlen(src) + 1);  // Copy with null
       terminator
    printf("Copied memory: %s\n", dest);
    return 0;
   }
   ```

2. `memmove`: Similar to `memcpy`, but handles overlapping memory regions safely.

   Function Format:

   ```c
   void *memmove(void *dest, const void *src, size_t n);
   ```

   Example:

   ```c
   int main() {
    char str[] = "Hello, McMaster!";
    memmove(str + 7, str, 5);  // Overlapping regions
    printf("Modified string: %s\n", str);
   ```

```
    return 0;
}
```

3. `memcmp`: Compares two memory blocks byte by byte.

   Function Format:

```
int memcmp(const void *ptr1, const void *ptr2, size_t n);
```

   Example:

```
int main() {
    char str1[] = "Hello";
    char str2[] = "McMaster";
    int result = memcmp(str1, str2, 5);
    printf("Memory comparison result: %d\n", result);
    return 0;
}
```

4. `memset`: Fills a block of memory with a specified value.

   Function Format:

```
void *memset(void *ptr, int value, size_t n);
```

   Example:

```
int main() {
    char str[50] = "Hello, McMaster!";
    memset(str, '*', 5);  // Replace first 5 characters
    printf("Modified string: %s\n", str);
    return 0;
}
```

Miscellaneous

1. `strdup` (Not part of the C standard, but widely available in POSIX-compliant systems): Creates a duplicate of a string using dynamically allocated memory.

   Function Format:

```
char *strdup(const char *str);
```

Example:

```c
#include <stdlib.h>

int main() {
 char str[] = "Hello, McMaster!";
 char *dup = strdup(str);
 if (dup) {
  printf("Duplicated string: %s\n", dup);
  free(dup);  // Don't forget to free memory!
 }
 return 0;
}
```

2. `strerror`:Returns a string that describes an error code.

   Function Format:

   ```c
   char *strerror(int errnum);
   ```

   Example:

   ```c
   #include <errno.h>

   int main() {
    FILE *file = fopen("nonexistent.txt", "r");
    if (!file) {
     printf("Error: %s\n", strerror(errno));
    }
    return 0;
   }
   ```

   Note: The above code needs an understanding about how reading a file works in C. Later in this course we'll discuss the topic.

These functions are essential tools for working with strings and memory in C. Remember to use them carefully to avoid issues like buffer overflows or memory leaks.