Character Strings (C string)

Learning Outcomes

After reading this section, you will be able to:

- Design data collections using arrays to manage information efficiently
- Stream data using standard library functions to interact with users

Introduction

Although some original programming languages focused on processing numerical information, most languages include extensive features for processing textual data. Textual data involves sets of characters. These sets are often referred to as character strings. The C language libraries provide facilities for processing character strings, treated as arrays of characters with a special delimiter.

This chapter introduces these C-style strings, highlights their distinguishing feature, and notes the advantage of using character strings to pass textual data from one function to another. This chapter includes the conversion specifiers for the input and output of character strings.

Definition (review)

A string is a `char` array with a special property which is a terminator element that follows the last ***meaningful character*** in the string. We refer to this terminator as the **null terminator** and identify it by the escape sequence `'\0'`.

char																	
																	\0

TERM DEFINITION

The term "Meaningful Characters" in these notes refers to the actual data content you want to manage in the C string character array.

The **null terminator** has the integral value of `0` on any host platform (in its collating sequence). All of its bits are 0's. The null terminator occupies the first position in the ASCII and EBCDIC.

The index identifying the null terminator element is the same as the number of meaningful characters in the string (including spaces between words).

char name																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**

HINT

The number of memory locations occupied by a C string (`char`) is one more than the number of meaningful characters in the string so as to hold the null terminator.

Allocating Memory

We allocate memory for a C string in the same way that we allocate memory for an array. Since the **null terminator** is one of the elements in the array, we must allocate memory for one extra character than the number of meaningful characters.

For example, to allocate memory for a string with up to `30` meaningful characters, we write:

```c
// Macro (placed after #includes) representing the intended meaningful number
//       of chars to manage in the char array
#define NAME_SIZE 30
.
.
// Variable declaration (char array) inside the function where you intend to use it
char name[NAME_SIZE + 1]; // 30 chars plus 1 char for the null terminator byte
```

Initializing Memory

To initialize a string at the time of memory allocation, we follow the definition with the assignment operator and the set of initial characters enclosed in braces.

```c
const char name[NAME_SIZE + 1] = {'M','y',' ','n','a','m','e',' ','i','s',' ',
                                  'A','r','n','o','l','d','\0'};
```

For a more compact form we enclosed the list of meaningful characters in double quotes.

```c
const char name[NAME_SIZE + 1] = "My name is Arnold";  // null-byte is automatically appended
```

The C compiler copies the characters in the string literal into the character string and appends the null-byte terminator after the last copied character.

char name																											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**	0	0	0	0	0	0	0	0	0	0

Since the number of initializers (18) is less than the number of elements (31) available, the compiler fills the uninitialized elements with 0's.

String Handling

Arrays of numbers require a separate variable to hold the number of elements that are filled. Unlike arrays of numbers, character strings do not require a separate variable for sizing. In iterations on the characters in a string, we check for the presence of the null terminator in our test conditions.

Iterations

The following program displays the string stored in name[31] character by character:

```c
// Iterations on character strings
// string_iterations.c

#include <stdio.h>

#define NAME_SIZE 30

int main(void)
{
    int i;
    const char name[NAME_SIZE + 1] = "My name is Arnold";
    for (i = 0; name[i] != '\0'; i++)
    {
        printf("%c", name[i]);
    }

    putchar('\n');

    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

Functions

Using a character string instead of an array of characters with a separate sizing variable achieves a more compact argument list for function calls. For example:

```
// Strings To Functions
// string_to_function.c

#include <stdio.h>

#define NAME_SIZE 30

void print(const char name[]);

int main(void)
{
    int i;
    const char name[NAME_SIZE + 1] = "My name is Arnold";

    print(name);
    return 0;
}

void print(const char name[])
{
    int i;

    for (i = 0; name[i] != '\0'; i++)
    {
        printf("%c", name[i]);
    }

    putchar('\n');
}
```

The above program produces the following output:

```
My name is Arnold
```

Formatted String Input

The `scanf()` and `fscanf()` library functions support conversion specifiers particularly designed for character string input. These specifiers are:

- `%s` - whitespace delimited set

- `%[]` - rule delimited set

The corresponding argument for these specifiers is the address of the string to be populated from the input stream.

%s

The `%s` conversion specifier

- reads all characters ***until*** the first whitespace character
- stores the characters read in the char array identified by the corresponding argument
- stores the null terminator in the char array after accepting the last character
- leaves the delimiting whitespace character and any subsequent characters in the input buffer

For example:

```
char name[NAME_SIZE + 1];
scanf("%s", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function will stop accepting input after the character `y` and stores the following:

char name																											
0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y	**\0**																									

The characters `' name is Arnold'` remain in the input buffer.

A qualifier on the conversion specifier limits the number of characters accepted. For instance, %10s reads no more than 10 characters:

```
char name[NAME_SIZE + 1];
scanf("%10s", name); // <=== User enters: Schwartzenegger
```

The `scanf()` function will stop accepting input after the character `n` and stores the following:

char name																											
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
S	c	h	w	a	r	t	z	e	n	**\0**																	

By specifying the maximum number of characters to be read at less than 31, we ensure that `scanf()` does not exceed the memory allocated for the string.

`%s` discards all leading whitespace characters.

For example, if enter many spaces before the Schwartzenegger value:

```
char name[NAME_SIZE + 1];
scanf("%10s", name);  // <=== User enters: '          Schwartzenegger'
```

Just as before, the `scanf()` function will stop accepting input after the character `n` but will also discard the leading spaces entered and stores the following:

char name																											
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
S	c	h	w	a	r	t	z	e	n	**\0**																	

Because `%s` discards ***leading whitespace***, it cannot accept an empty string; that is, `%s` does treat a `'\n'` in an empty input buffer as an empty string. If the buffer only contains `'\n'`, `scanf("%10s", name)` discards the `'\n'` and waits for non-whitespace input followed by another `'\n'`.

%[]

The `%[]` conversion specifier accepts input consisting only of a set of pre-selected characters. The brackets contain the admissible and/or inadmissible characters. The symbol `^` prefaces the list of **inadmissible** characters. The symbol `-` identifies a range of characters in an **inclusive** set.

For example, the `%[^\n]` conversion specifier:

- reads all characters until the newline ('\n')
- stores the characters read in the char array identified by the corresponding argument
- stores the null terminator in the char array after accepting the last character
- leaves the delimiting character (`'\n'`) in the input buffer

For example:

```
char name[NAME_SIZE + 1];
scanf("%[^\n]", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function accepts the full line an stores:

char name																											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**										

A qualifier on this conversion specifier before the opening bracket limits the number of characters accepted. For instance, `%10[^\n]` reads no more than 10 characters:

```
char name[NAME_SIZE + 1];
scanf("%10[^\n]", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function will store:

char name																											
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s	**\0**																	

We specify the maximum number of characters as the qualifier to ensure that `scanf()` does not store more characters than the allocated memory for the array size.

`%[ ]`, like `%s`, ignores any leading whitespace characters.

For example:

```
char name[NAME_SIZE + 1];
scanf("%10[^\n]", name);  // <=== User enters: '          My name is Arnold'
```

The `scanf()` function will store:

char name																											
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s	**\0**																	

Caution

Because `%[ ]` ignores leading whitespace, it cannot accept an empty string; that is, `%[^\n]` does treat a `'\n'` in an empty input buffer as an empty string. If the input buffer only contains `'\n'`, `scanf("%[^\n]", name)`, unlike `%s`, returns `0` and leaves `name` unchanged.

Example:

Consider a text file named spring.dat that contains:

```
Light Jacket
Long-Sleeved Shirts
Large Skateboards
```

The following program reads and displays this data:

```
// Reading from a file
// readFromFile.c

#include <stdio.h>

#define PHRASE_SIZE 60

int main(void)
{
    FILE *fp = NULL;
    char phrase[PHRASE_SIZE + 1];

    fp = fopen("spring.dat","r");

    if (fp != NULL)
    {
        while (fscanf(fp, "%60[^\n]%*c", phrase) != EOF)
        printf("%s\n", phrase);

        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

The above program produces the following output:

```
Light Jacket
Long-Sleeved Shirts
Large Skateboards
```

String Output

Formatted Output

The `printf()` and `fprintf()` library functions support the `%s` conversion specifier for character string output. The corresponding argument is the **address** of the character string or strings literal. Under this specifier `printf()` displays all of the characters from the address provided up to but **excluding** the null terminator byte. For example:

```
// Displaying Strings
// displayStrings.c

#include <stdio.h>

#define NAME_SIZE 30

int main(void)
{
    const char name[NAME_SIZE + 1] = "My name is Arnold";

    printf("%s\n", name);

    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

```
// Writing to a File
// writeToFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    const char phrase[] = "My name is Arnold";

    fp = fopen("alpha.txt","w");

    if (fp != NULL)
    {
        fprintf(fp, "%s\n", phrase);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

Qualifiers

Qualifiers on the `%s` specifier add detail control:

- `%20s` displays a string **right**-justified in a field of **20**
- `%-20s` displays a string **left**-justified in a field of **20**
- `%20.10s` displays the ***first*** **10** characters of a string **right**-justified in a field of **20**
- `%-20.10s` displays the ***first*** **10** characters of a string **left**-justified in a field of **20**

Unformatted Output

The `puts()` and `fputs()` library functions output a character string to the standard or specified output device respectively.

puts

The prototype for puts() is:

```
int puts(const char *);
```

The parameter receives the address of the character string to be displayed. For example:

```
// Displaying Lines
// puts.c

#include <stdio.h>

#define NAME_SIZE 30

int main(void)
{
    const char name[NAME_SIZE + 1] = "My name is Arnold";

    puts(name);

    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

fputs

`fputs()` writes a null-terminated string to a file. The prototype for `fputs()` is:

```
int fputs(const char *str, FILE *fp);
```

`str` receives the address of the string to be written and `fp` receives the address of the `FILE` object. `fputs()` returns a non-negative value if successful; `EOF` in the event of an error.