# Strings

Literal Strings and Pointers

- Up until now, we have only used *literal strings*; characters surrounded by double quotes; we have used them with `printf`:

  ```
  printf("Hello, world\n");
  printf("The value of i is %i\n", i);
  printf("The area is %f\n", PI * radius * radius);
  ```

- We have not assigned a string to a variable, nor have we entered strings via `scanf`.
- C doesn't have a built-in string type, per se.
- Strings, which are simply arrays of `char`, are used to implement strings.
- There are many library functions specifically designed to handle character arrays (strings) easily and efficiently
- Both C and C++ support these types of strings.
- They are commonly called NULL terminated strings or C-style strings to distinguish them from the more recent `std::string` used in C++.
- The last character in the string must be a NULL character, which is simply the value zero. (Don't confuse this with the NULL pointer, they are different)

There is a subtle difference between a string and an array of characters. This is how the first literal string above would be laid out in memory:

Literal strings are much like character arrays in that they can be used with pointers. In this example, `p` is a *char pointer* or *pointer to char* and it points to the first element in the string:

```
char *p = "Hello, world\n";
```

Visually:

We can print the string just as if it was a literal string:

```
printf(p);
```

Using the `%s` format specifier to print strings:

```
char *ph = "Hello";
char *pw = "world";
printf("%s, %s\n", ph, pw);
```

These three strings would look something like this (not necessarily adjacent in memory):

The terminating NULL (zero) character is very important when treating the array as a string:

```
char *ph = "Hello";
char w[] = {'H', 'e', 'l', 'l', 'o'};

printf("%s\n", ph); /* OK, a string      */
printf("%s\n", w);  /* Bad, not a string */
```

Output:

```
Hello
Hello�����������<@B
```

Another attempt:

```
  /* Manually add the terminator to the array */
char w[] = {'H', 'e', 'l', 'l', 'o', 0};

  /* Ok, now it's a string */
printf("%s\n", w);
```

We could print strings "the hard way", by printing one character at a time:

```
char *p = "Hello, world\n";
while (*p != 0)
  printf("%c", *p++);
```

After initilization:

After the **`while`** loop:

Make sure that you fully understand the difference between the pointer and the value that the pointer is pointing to:

```c
char *p = "Hello, world\n";

  /* This is the correct condition */
while (*p != 0)
  printf("%c", *p++);
```

```c
char *p = "Hello, world\n";

  /* INCORRECT */
while (p != 0)
  printf("%c", *p++);
```

Output from the incorrect code: (using gcc)

```
Hello, world
 %c Hello world %s, %s
 %s
 The value of i is %i
                                                                                                       ?
 a                                                                                                     @
@          hA  x@  l@          xA  �@                          �@  ,@  E@  O@  �@  ?A  ?A  ?A  $A      0
A           I a`y a�%   a?~?a�S a&+     a,/     aZ1     aN5     a        I
Y|    5 __main    F?_impure_ptr   �?calloc    �?cygwin_internal    ??dll_crt0__FP11per_process e?free  K?
malloc    >?printf     ?realloc   O?GetModuleHandleA    @   @   @   @   @   @   @   @   @  cygwin1.dll �@
  KERNEL32.dll
118871 [main] a 1808 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
 118871 [main] a 1808 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
 119867 [main] a 1808 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
 119867 [main] a 1808 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
 810735 [main] a 1808 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
 841004 [main] a 1808 _cygtls::handle_exceptions: Error while dumping state (probably corrupted stack)
```

The output before it crashed and burned when using Microsoft's compiler.

String Variables and Initialization

Initialization with character arrays:

```c
char s1[] = {'H', 'e', 'l', 'l', 'o'};    /* array of 5 chars */
char s2[] = {'H', 'e', 'l', 'l', 'o', 0}; /* array of 6 chars */
```

Initializing with strings:

```c
char s3[] = "Hello"; /* array of 6 chars; 5 + terminator                         */
char *s4 = "Hello";  /* pointer to a char; 6 chars in the "string"; 5 + terminator */
```

What is **sizeof** `s1, s2, s3, s4`? (Hint: What are the types?)

Initializing with fewer characters:

```c
char s5[10] = {'H', 'e', 'l', 'l', 'o'};    /* array of 10 chars, 5 characters are 0 */
char s6[8] = "Hello";                       /* array of 8 chars; 3 characters are 0  */
```

Given these declarations:

```c
char s[5]; /* array of 5 chars   */
char *p;   /* pointer to a char */
```

Use a loop to set each character and then print them out:

```c
  /* Set each character to A - E */
for (i = 0; i < 5; i++)
  s[i] = i + 'A';
```

```c
  /* Print out the characters: ABCDE */
for (i = 0; i < 5; i++)
  printf("%c", s[i]);
printf("\n");
```

A different loop doing the same thing (see an ASCII chart):

```c
  /* Set each character to A - E */
for (c = 'A'; c < 'A' + 5; c++)
  s[c - 'A'] = c;
```

```c
  /* Print out the characters: ABCDE */
for (i = 0; i < 5; i++)
  printf("%c", *(s + i));
```

Do something similar with p:

```c
  /* Print out the character that p points to */
printf("%c", p[0]);
printf("%c", *p);
```

You may get garbage, or it may crash:

```
     65 [main] a 2020 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
  22906 [main] a 2020 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
     65 [main] a 2020 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
  22906 [main] a 2020 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
 686199 [main] a 2020 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
 707734 [main] a 2020 _cygtls::handle_exceptions: Error while dumping state (probably corrupted stack)
```

Set p to point at something first:

```c
  /* Point p at s */
p = s;
```

Now print out the value:

```c
  /* Print out the character that p points to */
printf("%c", p[0]);
printf("%c", *p);
```

In a loop, print out all the characters that p points to: ABCDE. These are both the same:

```c
for (i = 0; i < 5; i++)
  printf("%c", p[i]);
```

```c
for (i = 0; i < 5; i++)
  printf("%c", *(p + i));
```

## String Input/Output

There's a convenient function for printing strings:

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

The `puts` function will print a newline automatically. Examples:

**Sample code**

```c
char *p1 = "Hello";
const char p2[] = "Hello";

puts("Hello");   /* literal string  */
puts(p1);        /* string variable */
puts(p2);        /* string variable */
puts("%s%i%d");  /* literal string  */
```

**Output**

```
Hello
Hello
Hello
%s%i%d
```

There's also a convenient function for printing a single character:

```c
int putchar(int c);
```

Example:

**Sample code**

```c
char c = 'H';
char *p = "ello";

putchar(c);       /* outputs one char, no newline */
while (*p)
  putchar(*p++); /* outputs one char, no newline */

putchar('\n');    /* print new line               */
```

**Output**

```
Hello
```

For input, we can us this:

```c
int gets(char *string);
```

Example:

```
char string[100]; /* 99 chars + 0 terminator */

puts("Type something: "); /* prompt the user */
gets(string);             /* read the string */
puts(string);             /* print it out    */
```

Output:

```
Type something:
I am not a great fool, so I can clearly not choose the wine in front of you.
I am not a great fool, so I can clearly not choose the wine in front of you.
```

- `gets` reads all characters until it encounters a newline character.
- The newline is read, then discarded and replaced with a 0 (so it's terminated).
- It is the programmer's (your!) responsibility to make sure that there is enough room to hold the input string.
- `gets` (and functions like it) is a very dangerous function and is likely responsible for many bugs/viruses/security problems today.
- A safer alternative is `fgets`.

We can also read a single character:

```
int getchar(void);
```

Example:

**Sample code**

```
int c = 0;

while (c != 'a')
{
  c = getchar(); /* read in a character   */
  putchar(c);    /* print out a character */
}
```

**Output**

```
This is a string (newline)
This is a (no newline)
```

## String Functions

Although strings are not truly built into the language, there are many functions specifically for dealing with NULL-terminated strings. You will need to include them:

```
#include <string.h>
```

Here are four of the more popular ones:

| Function Prototype | Description |
|---|---|
| `size_t`<br>`strlen(const char *string);` | Returns the length of the string, which is the number of characters int the string. It does not include the terminating 0. |
| `char*`<br>`strcpy(char *destination, const char *source);` | Copies the string pointed to by source into the string pointed to by destination. Destination must have enough space to hold the string from source. The return is destination. |
| `char *`<br>`strcat(char *destination, const char *source);` | Concatenates (joins) two strings by appending the string in source to the end of the string in destination. Destination must have enough space to accomodate both strings. The return is destination. |
| `int`<br>`strcmp(const char *s1, const char *s2);` | Compares two strings lexicographically (i.e. alphabetically). If string1 is less than string2, the return value is negative. If string1 is greater than string2, then the return value is positive. Otherwise the return is 0 (they are the same.) |