

String definition

- A sequence of characters handled as a unit
- In C, a string is an array of characters with final character equal to the “null character”, `\0`, also called the “null terminator”

String declaration

- Declaring a string:

```
char day[] = "monday";
```

// alternatively

```
const char *day_ptr = "monday";
```

- First declaration shows a string is like an *array*
- Second shows a string is like a *pointer* (more on this later)

String initialization

- Array of characters with final character equal to the “null character” \0

// this definition:

```
char day1[] = "monday";
```

// is the same as this:

```
char day2[] = {'m', 'o', 'n', 'd', 'a', 'y', '\0'};
```

- Note that both strings are *null-terminated*

String character access

- Access elements of the string using *square bracket* notation (a.k.a. *indexing*)

```
string_indexing_1.c:
```

```
#include <stdio.h>
```

```
//show how to access individual chars in a string
```

```
int main() {
```

```
    const char str[] = "hello";
```

```
    printf("%c %c %c\n", str[1], str[2], str[4]);
```

```
    return 0;
```

```
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra string_indexing_1.c
```

```
$ ./a.out
```

```
e l o
```

String copy bad

string_copy_1.c:

```
#include <stdio.h>
```

```
int main() {
```

```
    const char str[] = "hello";
```

```
    char str_copy[5];
```

```
    for(int i = 0; i < 5; i++) {
```

```
        str_copy[i] = str[i];
```

```
    }
```

```
    printf("%s\n", str); //use %s as string format specifier
```

```
    printf("%s\n", str_copy);
```

```
    return 0;
```

```
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra string_copy_1.c
```

```
$ ./a.out > junk  redirect the output into a file named junk
```

String copy good

```
string_copy_2.c:
#include <stdio.h>

int main() {
    const char str[] = "hello";
    char str_copy[6];
    for(int i = 0; i < 6; i++) {
        str_copy[i] = str[i];
    }
    printf("%s\n", str);    //use %s as string format specifier
    printf("%s\n", str_copy);
    return 0;    ↗ format string for printing c strings
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra string_copy_2.c
$ ./a.out
hello
hello
```

String null character positioning

- Strings are arrays of null-terminated (`\0`) characters
 - Null termination is used to indicate where the string ends

```
strlen_eg1.c:
#include <stdio.h>
#include <string.h> //include string.h for strlen
int main() {
    char s[] = "goodbye";
    printf("s = %s\n", s);
    s[4] = '\0'; //replace b with '\0'
    printf("But now, s = %s", s); //now only prints chars
                                //up to the (first) '\0'
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra strlen_eg1.c
$ ./a.out
s = goodbye
But now, s = good
```


String sizes

- Two size-related functions
 - `strlen` function returns number of chars before `\0`
 - `sizeof` function returns amount of space occupied by variable
 - both functions return unsigned long - `%lu` format string

strlen_eg2.c:

```
#include <stdio.h>
#include <string.h> //include string.h for strlen
int main() {
    char s[] = "goodbye";
    printf("s = %s, strlen(%s) = %lu\n", s, s, strlen(s));
    printf("s = %s, sizeof(%s) = %lu\n", s, s, sizeof(s));
    return 0;
}
```

format string for long unsigned



```
$ gcc -std=c99 -pedantic -Wall -Wextra strlen_eg2.c
```

```
$ ./a.out
```

```
s = goodbye, strlen(goodbye) = 7
```

```
s = goodbye, sizeof(goodbye) = 8
```


String sizes, moving null terminator

strlen_eg3.c:

```
#include <stdio.h>
#include <string.h>
int main() {
    char s[] = "goodbye";
    printf("s = %s, strlen(%s) = %lu\n", s, s, strlen(s));
    printf("s = %s, sizeof(%s) = %lu\n", s, s, sizeof(s));
    s[4] = '\0';
    printf("s = %s, strlen(%s) = %lu\n", s, s, strlen(s));
    printf("s = %s, sizeof(%s) = %lu\n", s, s, sizeof(s));
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra strlen_eg3.c
```

```
$ ./a.out
```

```
s = goodbye, strlen(goodbye) = 7
```

```
s = goodbye, sizeof(goodbye) = 8
```

```
s = good, strlen(good) = 4
```

```
s = good, sizeof(good) = 8
```

More sizeof details

- `sizeof(variable)` returns the total # of bytes occupied by variable
- `sizeof(type_name)` can be used also
- `char` type is one byte, so if `s` is a `char` array type, then `sizeof(s)` tells you the capacity of that array
- In general for an array: `sizeof(array_var) / sizeof(base_type)` tells you its declared size (number of elements it can hold)

sizeof examples

sizeof_eg.c:

```
#include <stdio.h>
int main() {
    char s[] = "goodbye";
    printf("sizeof(s) = %lu, sizeof(s[0]) = %lu\n",
           sizeof(s), sizeof(s[0]));
    int ra[] = {1, 2, 3, 4, 5};
    printf("sizeof(ra) = %lu, sizeof(int) = %lu\n",
           sizeof(ra), sizeof(int));
    printf("capacity of ra = %lu\n", sizeof(ra) / sizeof(int));
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra sizeof_eg.c
```

```
$ ./a.out
```

```
sizeof(s) = 8, sizeof(s[0]) = 1
```

```
sizeof(ra) = 20, sizeof(int) = 4
```

```
capacity of ra = 5
```

String operations - NOT

- no concatenation operator '+'
- no assignment '=' between strings declared as arrays - you can't do a whole assignment into an array because it is a fixed memory address
- we will have assignment between strings declared as pointers in the future

String library functions to the rescue

- `#include <string.h>` for helpful string functions:
 - `strlen(s)` returns length of string `s`, not including the `\0`
 - `strcmp(s1, s2)` compares two strings according to character ASCII values
 - negative: `s1` before `s2`
 - zero: `s1` and `s2` equal
 - positive: `s1` after `s2`

`strcmp("abc", "abd") ==> negative`
`strcmp("bg", "bg") ==> zero`
`strcmp("ad", "acg") ==> positive`
 - `strcpy(s1, s2)` copy effect is like `s1 = s2`
 - `s1` must be declared with a sufficient size
 - `strcat(s1, s2)` concatenate effect is like `s1 = s1 + s2`
 - `s1` must be declared with a sufficient size
 - See also: `strncpy`, `strncat`, `strtok`, others
 - <http://www.cplusplus.com/reference/cstring/>

Compiling to other than a.out

- When we compile a C program we can change the name of the output (executable) file from the default *a.out* name using the `-o` output flag followed by name of executable file `gcc source_file.c -o executable`
 - You could name your executable files `*.exe`, but it's not necessary in a unix environment
- The `-o` flag can be combined with all our other options as well
- The position of the `-o` flag and subsequent executable filename can be elsewhere, but we strongly recommend putting them at the end to avoid mixing up your executable and source file names.
 - **DANGER: mixing them up can overwrite your source code file!**

```
gcc -std=c99 -pedantic -Wall -Wextra my_program.c -o my_program
```