The Character String Terminator

In C, we declare a character string like so:

```c
char my_name[my_name_length+1];
```

Notice that a character string is declared exactly like a char array; in fact, a character string is a char array.

The only difference in the declaration is that the length of the array of char elements that represents the char string is one greater than the length of the string.

The last character in any C character string is the null character, called NUL, which corresponds to integer value 0:

```
'\0'
```

Thus, the null character (integer 0) is often referred to as the character string terminator.

In general, a numeric value that is used to indicate that a particular state has been reached — for example, the end of a list — is called a sentinel value.

So, the character string terminator NUL is a sentinel that indicates the end of the string in question.
How String Printing Really Works

The programs below behave identically.

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

int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];

    strcpy(my_name, "Henry Neeman");
    printf("My name is %s.\n", my_name);
    return 0;
}
```

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

int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];

    while (my_name[index] != '\0')
    {
        printf("%c", my_name[index]);
        index++;
    }
    printf("\n");
    return 0;
}
```

String Copy Function: `strcpy`

The C standard library function `strcpy` copies a string into a char array:

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

int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];
    char my_name2[my_name_length + 1];

    strcpy(my_name, "Henry Neeman");
    printf("My name is %s.\n", my_name);
    strcpy(my_name2, my_name);
    printf("My name is %s.\n", my_name2);
    return 0;
}
```

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

int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];

    my_name = "Henry Neeman"; /* THIS WON'T WORK! */
    return 0;
}
```
The String Placeholder

In a printf statement, the placeholder for a character string is:

```%s```

```% cat charstrcpy.c```
```#include <stdio.h>```
```#include <string.h>```
```int main ()```
```{ /* main */```
```    const int my_name_length = 12;```
```    char my_name[my_name_length + 1];```
```    char my_name2[my_name_length + 1];```
```    strcpy(my_name, "Henry Neeman");```
```    printf("My name is %s.\n", my_name);```
```    strcpy(my_name2, my_name);```
```    printf("My name is %s.\n", my_name2);```
```    return 0;```
```} /* main */```
```% gcc -o charstrcpy charstrcpy.c```
```% charstrcpy```
```My name is Henry Neeman.```
```My name is Henry Neeman.```
```

The strlen Function

The C standard library function strlen returns the length of the string that is passed to it, EXCLUDING THE STRING TERMINATOR:

```% cat charstrlen.c```
```#include <stdio.h>```
```#include <string.h>```
```int main ()```
```{ /* main */```
```    printf("strlen(%cHenry Neeman%c) = %d\n", '\042', '\042', strlen("Henry Neeman"));```
```    return 0;```
```} /* main */```
```% gcc -o charstrlen charstrlen.c```
```% charstrlen```
```strlen("Henry Neeman") = 12```
**Dynamic Allocation of Strings**

You can dynamically allocate the space for a string, just as you can for any other array:

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

int main ()
{
    char* my_name = (char*)NULL;
    int my_name_length;

    my_name_length = strlen("Henry Neeman");
    my_name = (char*)malloc(sizeof(char) * (my_name_length + 1));
    if (my_name == (char*)NULL) {
        printf("ERROR: can't allocate my_name.
        exit(-1);
    } /* if (my_name == (char*)NULL) */
    strcpy(my_name, "Henry Neeman");
    printf("My name is %s, my_name);
    free(my_name);
    my_name = (char*)NULL;
    return 0;
} /* main */
```

```bash
gcc -o charstrdyn charstrdyn.c
```

'My name is Henry Neeman.'

---

**Passing a String as a Function Argument**

Passing a string to a function is just like passing any other kind of array argument, whether statically allocated or dynamically allocated:

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

int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];
    char* my_name2 = (char*)NULL;
    void print_a_string(char* the_string);

    strcpy(my_name, "Henry Neeman");
    printf("My name is %s, my_name);
    print_a_string(my_name);
    my_name2 = (char*)malloc(sizeof(char) * (strlen(my_name) + 1));
    if (my_name2 == (char*)NULL) {
        printf("ERROR: can't allocate my_name2.
        exit(-1);
    } /* if (my_name2 == (char*)NULL) */
    strcpy(my_name2, my_name);
    printf("My name is still %s, my_name);
    print_a_string(my_name2);
    free(my_name2);
    my_name2 = (char*)NULL;
    return 0;
} /* main */

void print_a_string (char* the_string)
{
    printf("The string that was passed is: %s\n");
    if (the_string == (char*)NULL) {
        printf("ERROR: can't print a non-existent string\n");
        printf(" in print_a_string.\n");
        exit(-1);
    } /* if (the_string == (char*)NULL) */
    printf("%s\n", the_string);
} /* print_a_string */
```

```bash
gcc -o charstrpass charstrpass.c
```

'My name is Henry Neeman.'

'The string that was passed is: Henry Neeman.'

'The string that was passed is: Henry Neeman.'
String Comparisons

Just as numeric values can be compared, so can string values.

However, strings aren’t scalars.

In C, two strings are defined to be equal if they have the exact same contents.

String comparison is **case sensitive**.

Thus, if two strings are identical, except that, in a single character, they differ by case — for example, an "H" for one string corresponds to an "h" for the other — then they will not be equal.

For example:

"Henry" is not equal to "henry"

In C, strings are compared using the `strcmp` function from the C standard library.

THE RELATIONAL OPERATORS CANNOT BE USED TO COMPARE STRINGS!

```c
int main ()
{
    const int my_name_length = 12;
    char my_name[my_name_length + 1];
    char my_name2[my_name_length + 1];
    char my_first_name[my_name_length + 1];
    char my_first_name_lower[my_name_length + 1];
    char my_last_name[my_name_length + 1];
    strcpy(my_name, "Henry Neeman");
    strcpy(my_name2, my_name);
    strcpy(my_first_name, "Henry");
    strcpy(my_first_name_lower, "henry");
    strcpy(my_last_name, "Neeman");
    printf("strcmp(%s,%s) = %2d\n", my_name, my_name2);
    printf("strcmp(%s,%s) = %2d\n", my_first_name, my_first_name_lower);
    return 0;
}
```

Notice that the return value for `strcmp` can be interpreted as:

- zero: the strings are equal
- negative: the first string is less
- positive: the second string is less