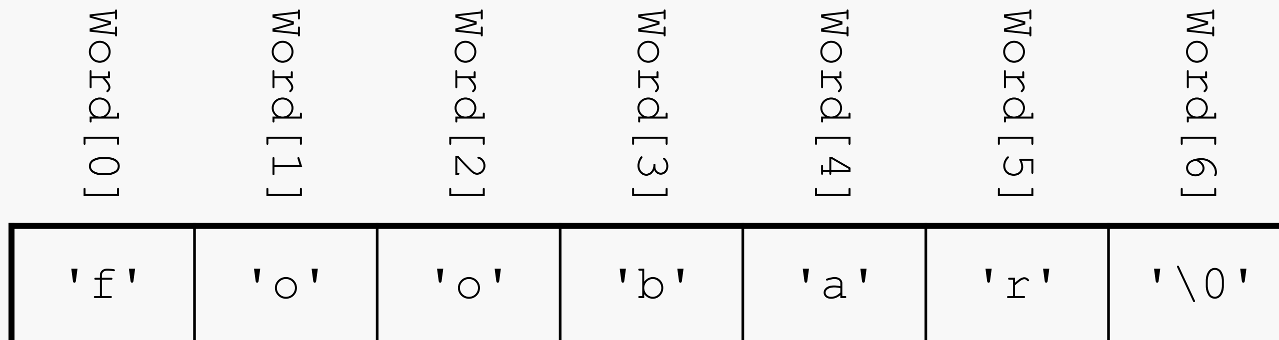
String Representation in C

There is no special type for (character) strings in C; rather, `char` arrays are used.

```
char Word[7]  = "foobar";
```



C treats char arrays as a special case in a number of ways.

If storing a character string (to use as a unit), you must ensure that a special character, the string terminator `'\0'` is stored in the first unused cell.

Failure to understand and abide by this is a frequent source of errors.

 ©2005-2015 McQuain

Some C String Library Functions

The C Standard Library includes a number of functions that support operations on memory and strings, including:

Copying:

```
void* memcpy(void* restrict s1, const void* restrict s2,
             size_t n);
```

Copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. Returns the value of s1.

```
char* strcpy(char* restrict s1, const char* restrict s2);
```

Copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. Returns the value of s1.

`string.h`

©2005-2015 McQuain

C String Library Hazards

The `memcpy()` and `strcpy()` functions illustrate classic hazards of the C library.

If the target of the parameter `s1` to `memcpy()` is smaller than `n` bytes, then `memcpy()` will attempt to write data past the end of the target, likely resulting in a logic error and possibly a runtime error. A similar issue arises with the target of `s2`.

The same issue arises with `strcpy()`, but `strcpy()` doesn't even take a parameter specifying the maximum number of bytes to be copied, so there is no way for `strcpy()` to even attempt to enforce any safety measures.

Worse, if the target of the parameter `s1` to `strcpy()` is not properly 0-terminated, then the `strcpy()` function will continue copying until a 0-byte is encountered, or until a runtime error occurs. Either way, the effect will not be good.

©2005-2015 McQuain

Safer Copying

For safer copying:

```
char* strncpy(char* restrict s1,const char* restrict s2,
              size_t n);
```

Copies not more than n characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.

If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by `s2` is a string that is shorter than `n` characters, null characters are appended to the copy in the array pointed to by `s1`, until `n` characters in all have been written.

Returns the value of `s1`.

(Of course, this raises the hazard of an unreported truncation if `s2` contains more than `n` characters that were to be copied to `s1`, and null termination of the destination is not guaranteed in that case.)

©2005-2015 McQuain

Another C String Library Function

Length:

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

Computes the length of the string pointed to by `s`.

Returns the number of characters that precede the terminating null character.

Hazard: if there's no terminating null character then `strlen()` will read until it encounters a null byte or a runtime error occurs.

 ©2005-2015 McQuain

Concatenation:

```
char* strcat(char* restrict s1, const char* restrict s2);
```

Appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`.

If copying takes place between objects that overlap, the behavior is undefined.

Returns the value of `s1`.

```
char* strncat(char* restrict s1, const char* restrict s2,
              size_t n);
```

Appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result.

If copying takes place between objects that overlap, the behavior is undefined.

Returns the value of `s1`.

©2005-2015 McQuain

More C String Library Functions

Comparison:

```
int strcmp(const char* s1, const char* s2);
```

Compares the string pointed to by `s1` to the string pointed to by `s2`.

The `strcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

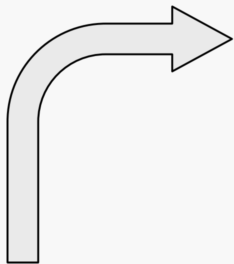
```
int strncmp(const char* s1, const char* s2, size_t n);
```

Compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

The `strncmp` function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

©2005-2015 McQuain

Bad strcpy()!



```
Ubu > gcc -o str03 -m32 -std=c99 -Wall str03.c
Ubu > str03
s1:    & R:   the C Programming Language
s2:  K & R:   the C Programming Language
```

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

int main() {

   char  s1[] = "K & R:  the C Programming Language";
   char  s2[1];

   strcpy(s2, s1);             // s2 is too small!

   printf("s1:  %s\n", s1);
   printf("s2:  %s\n", s2);

   return 0;
}
```

 ©2005-2015 McQuain

x86-64 Aside

BTW, here's what happened when the same code was compiled for a 64-bit target:

```
Ubu > gcc -o str03_64 -std=c99 -Wall str03.c
Ubu > str03_64
s1:   & R:  the C Prrogramming Language
s2:  K& R:  the C Prrogramming Language
```

There are no profound lessons here, but note that the behavior is interestingly different.

When you're debugging, it may be useful to know whether you have a binary for a 32- or 64-bit environment.

 ©2005-2015 McQuain

The Devil's Function

The C language included the regrettable function:

```
char* gets(char* s);
```

The intent was to provide a method for reading character data from standard input to a `char` array.

The obvious flaw is the omission of any indication to `gets()` as to the size of the buffer pointed to by the parameter `s`.

Imagine what might happen if the buffer was far too small.

Imagine what might happen if the buffer was on the stack.

The function is officially deprecated, but it is still provided by `gcc` and on Linux systems.

©2005-2015 McQuain

Some Historical Perspective

There's an interesting recent column, by Poul-Henning Kamp, on the costs and consequences of the decision to use null-terminated arrays to represent strings in C (and other languages influenced by the design of C):

> . . .
>
> Should the C language represent strings as an address + length tuple or just as the address with a magic character (NUL) marking the end? This is a decision that the dynamic trio of Ken Thompson, Dennis Ritchie, and Brian Kernighan must have made one day in the early 1970s, and they had full freedom to choose either way. I have not found any record of the decision, which I admit is a weak point in its candidacy: I do not have proof that it was a conscious decision.

 ©2005-2015 McQuain

Some Historical Perspective

> As far as I can determine from my research, however, the address + length format was preferred by the majority of programming languages at the time, whereas the address + magic_marker format was used mostly in assembly programs. As the C language was a development from assembly to a portable high-level language, I have a hard time believing that Ken, Dennis, and Brian gave it no thought at all.
>
> Using an address + length format would cost one more byte of overhead than an address + magic_marker format, and their PDP computer had limited core memory. In other words, this could have been a perfectly typical and rational IT or CS decision, like the many similar decisions we all make every day; but this one had quite atypical economic consequences.
>
> . . .

http://queue.acm.org/detail.cfm?id=2010365

©2005-2015 McQuain