String Representation in C

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

```c
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.
Issues with String Termination

When a char array is initialized at the point of declaration, a string terminator is added by the compiler (as long as you provide sufficient room):

```c
char Word[7] = "foobar";
printf("%s", Word); // writes "foobar"
```

Otherwise, learn to be careful:

```c
int main() {
    char Word[7] = "foobar";
    printf("%s
", Word);

    char Term[6] = "foobar";
    printf("%s
", Term);

    char Hmmm[6] = {'f', 'o', 'o', 'b', 'a', 'r'};
    printf("%s
", Hmmm);

    char Hooo[7] = {'f', 'o', 'o', 'b', 'a', 'r'};
    printf("%s
", Hooo);

    return 0;
}
```
Stack Layout

+28   0
 27    r
 26    a
 25    b
+24    o
 23    o
 22    f     Term
 21    r
+20    a
 19    b
 18    o
 17    o
+16    f     Hmm
 ...  

+28   0
 27    r
 26    a
 25    b
+24    o
 23    o
 22    f     Term
 21    r
+20    a
 19    b
 18    o
 17    o
+16    f     Hmm
 ...  

+15    0
 14    r
 13    a
+12    b
 11    o
 10    o
  9    f     Word
+ 8    0
  7    r
  6    a
  5    b
+ 4    o
  3    o
  2    f     Hooo
  1    0
 esp
Another Example

VERY careful:

```c
int main() {
    char Term[6] = "foobar";
    printf("%s\n", Term);
    return 0;
}
```

Note: YMMV with the output… this will very possibly not be the same for you.

Remember this when we discuss the run-time stack in detail later.
Just to be perverse...

The following is legal:

```c
int main() {
    char Term[] = "foobar"; // compiler allocates correct space
    printf("%s\n", Term);
    return 0;
}
```

This is not terribly useful… but you should know about it.
Some C String Library Functions

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

Copying:

```c
size_t 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.

```c
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.
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.
For safer copying:

```c
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 \texttt{s2} to the array pointed to by \texttt{s1}.

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

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

Returns the value of \texttt{s1}.

(Of course, this raises the hazard of an unreported truncation if \texttt{s2} contains more than \texttt{n} characters that were to be copied to \texttt{s1}, and null termination of the destination is not guaranteed.)
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.
More C String Library Functions

Concatenation:

```c
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 \( s1 \).

```c
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 \( s1 \).
Comparison:

```c
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`.

```c
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`. 
Illustrate:

str03 example showing effect/lack thereof of size-related errors in the destination

what happens with same example if we have no terminator on source array?

get into buffer overflow issues and raise specter of this happening on the stack; what might be at risk?
ISO Standard Alternative

Some alternatives have been proposed, with alleged safety advantages:

```c
errno_t strcpy_s(char* restrict s1, rsize_t s1max,
                 const char* restrict s2);
```

**Runtime constraints:**

Neither `s1` nor `s2` shall be a null pointer.
`s1max` shall not be greater than `RSIZE_MAX`.
`s1max` shall not equal zero.
`s1max` shall be greater than `strnlen_s(s2, s1max)`.

Copying shall not take place between objects that overlap.

If there is a runtime-constraint violation, then if `s1` is not a null pointer and 
`s1max` is greater than zero and not greater than `RSIZE_MAX`, then `strcpy_s` 
sets `s1[0]` to the null character.

This is not supported by gcc, nor is it likely to be in the future.
The OpenBSD movement proposed another alternative:

```c
size_t strlcpy(char* s1, const char* s2, size_t n);
```

Similar to `strncpy()` but truncates, if necessary, to ensure null-termination of the copy.

This is not supported by `gcc`, nor is it likely to be in the future.
The Devil's Function

The C language included the regrettable function:

```c
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.

See:

Some Historical Perspective

There's an interesting recent column 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):

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