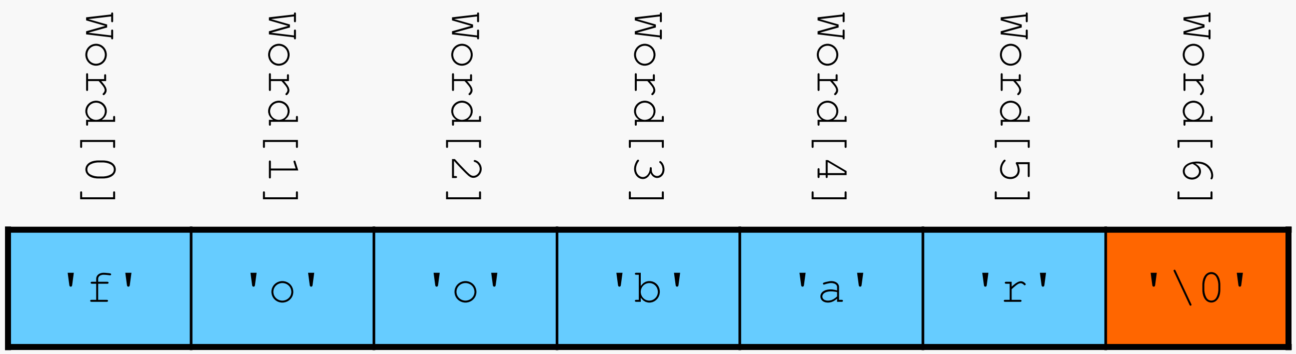
String Representation in C

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.

©2005-2013 McQuain

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`

Whatever perspective we take on the original decision, we must deal with it.

©2005-2013 McQuain

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):

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

printf("%s", Word);                    // writes "foobar"
```

Otherwise, learn to be careful:

```
int main() {

   char Word[7] = "foobar";
   printf("%s\n", Word);

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

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

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

   return 0;
}
```

```
foobar
foobar
foobarfoobar
foobar
```


©2005-2013 McQuain

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      Hmmm
     . . .
```

```
      . . .
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
```

This is only one possible stack layout for the data… nothing is guaranteed aside from the fact that storage for an array is always allocated contiguously.

Another Example

VERY careful:

```
int main() {

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

   return 0;
}
```

```
foobar
```

```
foobarDI"
```

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

The effect of errors like this is difficult to predict; you must learn to avoid them.

©2005-2013 McQuain

string.h: Memory and String Functions

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

Length:
```
size_t strlen(const char* s1);
```

Copying:
```
size_t memcpy(void* restrict s1, const void* restrict s2, size_t n);
char*  strcpy(char* restrict s1, const char* restrict s2);
char*  strncpy(char* restrict s1,const char* restrict s2, size_t n);
```

Comparing:
```
int    memcmp(const void* s1, const void* s2, size_t n);
int    strcmp(const char* s1, const char* s2);
int    strncmp(const char* s1, const char* s2, size_t n);
```

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


©2005-2013 McQuain

string.h: Copy Functions

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

Copying:

```
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.

```
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.

©2005-2013 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.

©2005-2013 McQuain

string.h: 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.)

©2005-2013 McQuain

string.h: Length Function

Length:

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

The `strlen()` function shall compute the number of bytes in the string to which *s* points, not including the terminating null byte.

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

©2005-2013 McQuain

string.h: Concatenation Functions

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`.

 ©2005-2013 McQuain

string.h: Comparison 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`.

©2005-2013 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.

See:

http://accu.informika.ru/acornsig/public/caugers/volume2/issue4/gets.html

©2005-2013 McQuain

Examples

The following slides contain some short examples illustrating the use of the C string functions in a small, practical scenario.

The input file being used consists of GIS (geographic information system) records; each record is stored on a single line, by itself, and consists of a sequence of fields, separated by pipe characters ('|'):

```
1674762|Tremont Estates|Populated Place|VA|51|Montgomery|121|371412N|0802601W|...|Blacksburg|11/13/1995|
1465730|Den Hill Cemetery|Cemetery|VA|51|Montgomery|121|370920N|0801844W|...|Ironto|09/28/1979|
1674497|Carma Heights|Populated Place|VA|51|Montgomery|121|370955N|0802613W|...|Blacksburg|11/13/1995|
1674655|Norris Hall|Building|VA|51|Montgomery|121|371348N|0802521W|...|Blacksburg|11/13/1995|
1498467|Christiansburg|Populated Place|VA|51|Montgomery|121|370747N|0802432W|...|Blacksburg|09/28/1979
```

The significance of the fields isn't important for us, but you can find out more at the website for the Geographic Names Information System (*nhd.usgs.gov/gnis.html*).

It is worth noting that some fields in some records may be empty.

In that case, there will be two successive pipe characters, with nothing separating them.

©2005-2013 McQuain

Reading a Line of Text

Here's a function to read a line of text. It reads to the end of the current line in the file, but will not put more than `limit` characters into the array, plus a terminator.

```
uint32_t readline(FILE* fp, char* line, uint32_t limit) {

   uint32_t status = 0;       // 0 = OK; 1 == excess data on line
   int ch;                    // character just read; fgetc() returns an int
   uint32_t nRead = 0;        // number of characters read so far

   // read until we reach a newline or EOF
   while ( !feof(fp) && (ch = fgetc(fp)) != '\n' ) {

      // see if the line is longer than the specified limit
      if ( nRead > limit )
         status = 1;

      // don't put more than limit characters into line
      if ( nRead < limit ) {
         line[nRead] = (char) ch;
         nRead++;
      }
   }
   line[nRead] = '\0';  // write terminator after last char in line
   return status;
}
```


©2005-2013 McQuain

Reading All the Lines in a File

Here's some code that uses `readline()` to read all the lines in the file.

```
// try to read a line
uint32_t status = readline(fp, line, MAXLEN);

// stop when reach end of input file
while ( !feof(fp) ) {

   // check for a short read
   if ( status == 1 )
      fprintf(stdout, "Excess data; did not read entire line!\n");

   // get length of current line
   len = strlen(line);

   // echo the current line
   fprintf(stdout, "Read %"PRIu32" characters: %s\n", len, line);

   // try to read another line
   status = readline(fp, line, MAXLEN);
}
```

The pattern here is intended to guarantee that we check for `EOF` immediately after each attempt to read a character; this is sufficiently robust for the present case, but it can be improved by also employing `ferror()`.

©2005-2013 McQuain

`fgetc()` and `feof()`

```
int fgetc(FILE* stream);
```

> Upon successful completion, `fgetc()` shall return the next byte from the input stream pointed to by `stream`.
>
> If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `fgetc()` shall return `EOF`.
>
> If a read error occurs, the error indicator for the stream shall be set, `fgetc()` shall return `EOF`, and shall set `errno` to indicate the error.

```
int feof(FILE* stream);
```

> The `feof()` function shall return non-zero if and only if the end-of-file indicator is set for `stream`.

Be very careful with this... it does not tell you whether you've reached the last character in the file, but whether you've tried to read beyond that character.

©2005-2013 McQuain

Tokenizing a Line

Here's some code that uses `strtok()` to extract all the fields in a GIS record.

```
uint32_t tokenize(FILE* fp, char* const str, const char* const delimiters) {

   if ( str == NULL || *str == '\0' ) return 0;

   uint32_t nTokens = 0;

   char* currToken = strtok(str, delimiters);          // prime the pump

   while ( currToken != NULL ) {      // strtok() returns NULL if no token

      nTokens++;

      if ( strlen(currToken) > 0 ) {
         fprintf(fp, "%5"PRIu32": %s\n", nTokens, currToken);
      }

      currToken = strtok(NULL, delimiters);
   }
   return nTokens;
}
```


©2005-2013 McQuain

`strtok()`

```
char* strtok(char* s, const char* sep);
```

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `s2`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `s2` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first byte that is *not* contained in the current separator string pointed to by `s2`. If no such byte is found, then there are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

©2005-2013 McQuain

`strtok()` Example


```
1674762|Tremont Estates|Populated Place|VA|51|Montgomery|121|...
```

```
char* t0 = strtok(p, "|");
```

```
1674762○Tremont Estates|Populated Place|VA|51|Montgomery|121|...
```

```
char* t1 = strtok(NULL, "|");
```

```
1674762○Tremont Estates○Populated Place|VA|51|Montgomery|121|...
```

??

```
char* t2 = strtok(NULL, "|");
```


©2005-2013 McQuain

Tokenizing a Line


```
... Same code as before to loop and read lines

// stop when reach end of input file
while ( !feof(fp) ) {

   ...

   if ( len > 0 ) {
      tokenize(stdout, line, "|");
   }

   // try to read another line
   status = readline(fp, line, MAX
}
```

```
Read 136 characters: 1674762|T...
     1: 1674762
     2: Tremont Estates
     3: Populated Place
     4: VA
     5: 51
     6: Montgomery
     7: 121
     8: 371412N
     9: 0802601W
    10: 37.2367952
    11: -80.4336623
    12: 641
    13: 2103
    14: Blacksburg
    15: 11/13/1995
```


©2005-2013 McQuain