

Dealing with text

Comments and questions to [John Rowe](#).

So far we have been dealing with character strings, (or just "strings" for short) enclosed inside double quotes ("") without really saying what they are.

As well as the fixed strings we have been dealing with, C allows us to deal with individual characters as well as modifiable strings via arrays of characters. C also provides a range of standard routines for inspecting and modifying characters and strings.

This lecture just deals with strings using the English character set. Should you wish to write a multi-lingual program you will need to investigate [unicode](#).

Single characters

C denotes **single characters** using **single quotes**, for example:

```
char mychar = 'a';
```

The use of single quotes for single characters is very important.

A char is a one-byte integer

Since the data in a computer's memory consist of zeros and ones, not letters, C stores characters by assigning a one-byte integer value to each character. When a character value is stored in memory it is this one-byte integer that is written. When library routines such as `printf()` read this one-byte integer from memory and are instructed to interpret it as a character they translate the integer value back to the appropriate printable character. Therefore, technically a `char` is a one-byte integer.

Even though internally C treats characters as a type of integer one byte long, we should never rely on any particular character having any particular integer value.

Single characters have the "`%c`" format specifier to `printf()` and `scanf()`. `%c` is unusual in that on input it does **not** skip white space by default. To read in a single character skipping white space put a space before the `%c`:

```
char mychar;

printf("Please enter a single character, spaces will be ignored\n");
scanf(" %c", &mychar);
```

Note the space before "`%c`".

Don't do this

In practice just about every modern machine uses a mapping called ASCII to decide which one-byte integer value represents which character. In ASCII the value of integer that is used internally to represent the character 'a' is 97. This can lead to code like this:

```
char mychar = 97; /* Bad! */
```

This is legal but has two problems:

1. It's not guaranteed. Nowhere does the C standard say compilers have to use this value.
2. It's not clear:

```
char mychar = 'a';
```

is much easier to understand.

Similarly:

```
if ( mychar == 97) { /* BAD!! */
    ...
}

if ( mychar == 'a') { /* Use this instead */
    ...
}
```

If we need to know which integer the computer uses to represent a character we are doing something wrong.

Have we mentioned the megaprinciple?

With this warning we shall use the ASCII character to integer encoding through out these notes on the clear understanding that our code must never depend on it. The only thing guaranteed is that **no character is represented by the value zero**, as this is used to indicate the end of a string of characters.

Special characters

As with character strings, single characters that are hard to represent inside single quotes are specified using the backslash, for example:

```
char quote = '\\';
char newline = '\\n';
char backslash = '\\\\';
```

Decimal digits

One mistake to avoid is to confuse the integer value used to internally represent a digit **character** with the integer whose value is that digit. For example the character '9' has the ASCII value 57 (the byte that gets stored in the computer's memory), where as the ASCII value 9 is used to represent the tab character.

Character strings

With its typical economy and simplicity C does not define a separate "string" type, instead it treats character strings as **arrays of characters**.

Reminder: one-dimensional arrays

- We can think of the bytes of the computer's memory as being numbered "1", "2", "3", etc.
- C stores the values of array elements in consecutive locations in memory.
- The "value" of the name of an array is just the value of the location of the start of the array. So if `x` is an array of floats starting at location 1000 then `x[0]` means the float stored starting at 1000, `x[1]` means the float stored starting at 1004, `x[n]` means the float stored starting at $1000+4*n$, etc.
- C does not keep track of the end of the array: we have to do it ourselves.
- If we go a bit over the end of an array we will change the values of random variables, if we go a lot over the end we are likely to crash the entire program.

Fixed character strings

We have already used fixed character strings enclosed within double quotes as arguments to `printf()`, `scanf()`, etc.

When the compiler encounters a fixed string such as "Hello, world\n", it creates a non-modifiable array of one-byte integers with the appropriate values and passes the address of that array to `printf()` in exactly the same way as any other array.

Then when `printf()` looks at the array it has been passed it will see the first byte is 72 ('H'), the second 101 ('e'), etc. This raises the question: "how does it know when it has got to the end?"

Remember, the computer's memory goes on way past the end of any particular array. The address of a variable tells us where the memory allocated for that array starts but not where it ends.

The answer, as alluded to above, is that C appends a stop-byte of value zero to the end of every character string. Thus for example the string "abc" takes up **four** bytes of storage, not three, the fourth byte being zero.

The integer constant zero can obviously be written simply as just 0, but when used in a character context it is conventional to write it in the form '\\0'. This is another example of using the backslash inside a character or string constant to indicate a special character, in this case a literal integer value. But either will work.

Strings have the "%s" format specifier to `printf()` and `scanf()`. For `scanf()` "%s" can only be used to read in character strings **without spaces**. We will see how to deal with this in a later lecture.

This is the only situation where C allows arrays without a name ("anonymous arrays")

The ASCII value for the character '0' is forty-eight, not zero.

Passing fixed strings to functions

As a string is just an array of one-byte integers it can be passed to a function (that is, its address can be passed to a function) just like any other array. There is one qualification however; the function must not try to modify the string!

On my machine the following code produces a segmentation fault (SIGSEGV):

```
#include <stdio.h>
/*
 * Demonstrate problem modifying a fixed string
 */
void myfunction (char message[]) {
    printf("The original message was: %s\n", message);

    message[0] = 'A';
    printf("The new message is: %s\n", message);
}

int main() {
```

```

myfunction("Hello, world");
return 0;
}

```

The const qualifier

When a function takes (the address of) an array as an argument but guarantees not to modify it, this can be indicated by adding the *qualifier* **const** (short for constant) in front of the argument declaration.

The following code demonstrates a simple function to read in an integer within a specified range with a helpful message. Since the message is only printed, not modified, we declare it as **const**.

```

#include <stdio.h>

/*
 * Read an integer in the range min to max inclusive
 */
int getint(const char message[], int min, int max) {
    while (1 == 1) {
        int input;

        printf("%s\n Please enter a number from %d to %d\n",
            message, min, max);
        scanf("%d", &input);

        if ( input < min)
            printf("Sorry, the minimum possible value is %d. ", min);
        else if ( input > max)
            printf("Sorry, the maximum possible value is %d. ", max);
        else
            return input;

        printf("Please try again\n\n");
    }
}

int main() {
    int value;

    value = getint("Pick a number!", 1, 10);
    printf("You chose: %d\n", value);
    return 0;
}

```

The **const** qualifier can be used for arrays of any type, not just integers. If the function had a prototype we would have to add the **const** qualifier there as well as the whole point about prototypes and function definitions is that they agree.

Non-fixed strings, or arrays of characters.

If we want character strings we can modify, we can declare them like any other array. As a further convenience we can use a string such as "abc" as an initialiser instead of { 'a', 'b', 'c', '\0' } (remember, the string requires **four** bytes because of the final zero).

The following example declares two four byte character arrays with one initialised and one left uninitialised:

```
char str[] = "abc", string2[4];
```

Each can be used to store a three-character string, plus the stop-byte of zero, and can be modified.

Like all arrays we cannot just write `string2 = str1` but there is a function called `strncpy()` to do it for us (see below).

Things to look out for

1. 'a' != "a"

The first is a single character, the second is the address of a two-byte quantity, the first byte having the value 'a', the second zero or '\0'.

The format for a single character is "%c", that of a string "%s"

2. '0' != '\0', '1' != 1

In both cases the left-hand value is the integer code for that character (probably 48 and 49 respectively), the second is

the integer zero or one.

3. Strings require one byte more to store than their length, arrays can store one fewer characters than their size

This is used to store the stop-byte, integer zero or '\0'.

Utility functions for characters and strings

C provides a number of useful functions for handling characters and strings.

Character-based functions: <ctype.h>

The include file <ctype.h> ("Character TYPE .h") provides a number of useful functions to test, and in two cases below convert, the type of a character. All take a single argument which is a character (not a character string).

"is...()" tests

Useful <ctype.h> functions			
Function	Description	Example (True)	Example (False)
isalpha()	Alphabetic (letter: includes '_')	isalpha('A') isalpha('_')	isalpha('7')
isupper()	Upper case letter	isupper('A')	isupper('a') isupper('?')
islower()	Lower case letter	islower('a')	islower('A') islower('?')
isdigit()	Decimal digit	isdigit('4')	isdigit('B')
isalnum()	Letter or decimal digit	isalnum('m') isalnum('9')	isalnum('?')
isspace()	Space	isspace(' ') isspace('\t')	isspace('G')
ispunct()	Punctuation	ispunct('?')	ispunct('w')

Example

The following function looks at a character string to count the number of letters, digits, punctuation characters and spaces.

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

/*
 * Count the number of letters, digits, punctuation
 * characters and spaces in a well-known phrase
 */
void countem(const char string[]) {
    int alphas = 0, digits = 0, spaces = 0, punct = 0, others = 0;
    int i;

    for(i = 0; string[i] != '\0'; ++i) {
        if (isalpha(string[i]))
            ++alphas;
        else if (isdigit(string[i]))
            ++digits;
        else if (isspace(string[i]))
            ++spaces;
        else if (ispunct(string[i]))
            ++punct;
        else
            ++others;
    }
}
```

```

printf("The string: %s\n"
      "contains %d letters, %d digits, %d spaces,\n"
      "%d punctuation characters and %d other characters\n",
      string, alphas, digits, spaces, punct, others);
}

int main() {
    countem("Hello, world");
    return 0;
}

```

You will note that the character string is declared as "const char" as the string itself is not modified.

The output is:

```

The string: Hello, world
contains 10 letters, 0 digits, 1 spaces,
1 punctuation characters and 0 other characters

```

toupper() and tolower()

The functions `toupper()` and `tolower()`, also defined in `ctype.h`, also take a single character as an argument and return it converted to upper or lower case. Usefully, if the argument is a non-letter, or is already the correct case, they just return the argument.

The following function translates its argument string to upper case.

```

#include <stdio.h>
#include <ctype.h>

/*
 * Convert a string to UPPER CASE
 */
void shout(char string[]) {
    int i;

    for(i = 0; string[i] != '\0'; ++i) {
        string[i] = toupper(string[i]);
    }
}

int main() {
    char helloworld[] = "Hello, world";

    printf("The original string is: %s\n", helloworld);
    shout(helloworld);
    printf("The upper-case string is: %s\n", helloworld);

    return 0;
}

```

Notice that we have dropped the "const" qualifier to the argument to `shout()` and we have had to copy our favourite phrase into a (modifiable) character array; calling `shout()` with a fixed string would have led to a segmentation error.

The output is:

```

The original string is: Hello, world
The upper-case string is: HELLO, WORLD

```

Notice how the 'H' and ', ' are unchanged.

String utility functions: <string.h>

The file `<string.h>` provides a number of functions for dealing with **strings**, not individual characters.

In the table below, all strings are guaranteed to be unmodified ("const") except those called *dest*.

Useful <string.h> functions	
Function	Description
<code>strlen(string)</code>	String length <code>strlen("abc") == 3</code>
<code>strncpy(dest, source, nbytes)</code>	String copy (See example and warning)
<code>strncat(dest, source, nbytes)</code>	Concatenate two strings
<code>strcmp(string1, string2)</code>	Compare two strings
<code>strstr(string1, string2)</code>	Find <i>string2</i> inside <i>string1</i> .

strlen()

Note that `strlen` returns the **length of the string**, not size of the array it is stored in. For example the code:

```
char string[12] = "abc";
printf("The length of \"%s\" is %d\n", string, strlen(string));
```

will print the value three, not twelve. Note too how we have used the backslash to put a double-quote character inside a quoted string.

strncpy()

The `strncpy(destination, source, n)` function copies at most `n` bytes of the source string to the destination. The syntax is supposed to be reminiscent of the not-allowed `destination = source` with a sanity check.

strncpy() warning

The `strncpy()` has a subtle flaw: although it correctly refuses to write over the end of the destination array (good), in the case that the length of the source is larger than `n` the final character written to the destination array is not zero, it is the `n`th byte of the source. This means that the destination does not have a terminating zero and hence anything that tries to treat it as a C string will run off the end.

One solution is to make the destination array one byte "too big" and initialise it all to zeros. That way the final byte of the destination is always zero, as in this example:

```
/*
 * Safer way to use strncpy() to handle overflows
 */
#define N 8
int main() {
    char buffer[N+1] = ""; /* One byte longer and initialised to all zeros */

    strncpy(buffer, "Hello, world", N);

    printf("The truncated string is: %s\n", buffer);
    return 0;
}
```

It's worth noting that we have used the fact that when we initialise an array with fewer elements that it needs the rest are all set to zero.

Of course, another way to solve the `strncpy()` problem is to always call `strncpy()` with argument `N-1`, or to write a "wrapper" function called, say `mystrncpy()`, that always writes a zero to the final byte..

strncat()

The `strncat()` function **catenates** (joins) the contents of the second argument onto the end of the first, the first character of the second argument over-writing the zero at the end of the first. For example:

```
char dest[64] = "Hello, ";
strncat(dest, "world", 12);
```

creates a modifiable copy of the text we all know and love. `strncat()` does not have the same flaw as `strncpy()`

in that if it runs out of space the final character is zero, but it should be remembered that the final argument is the maximum number of bytes to be copied (appended or concatenated), not the maximum final length of the resultant string. Try this (assuming N is large enough for the initial string):

```
char dest[N] = "And I think to myself, ";
...
strncat(dest, "Hello, world", N - strlen(dest));
```

Both `strncpy()` and `strncat()` have versions without the 'n' and the final maximum-length argument, `strcpy()` and `strcat()`. These are best avoided.

strcmp() and strstr()

The `strcmp(str1, str2)` function returns the value zero if `str1` and `str2` are the same, a (strictly) negative integer if `str1` comes before `str2` in the alphabet and a (strictly) positive integer if `str1` comes after `str2` in the alphabet. The classic use is to check for equality by seeing if `strcmp()` returns zero.

As always with arrays we cannot test for string equality using `str1 == str2` as this tests to see if the **addresses** of the two strings are the same, i.e. they both refer to the same character array, rather than two character arrays which contain the same string.

The function `strstr(str1, str2)` tells us if `str1` contains the string `str2`. (The return value of `strstr()` can also tell us where in `str1` `str2` appears.)

Here is an example:

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

/*
 * Simple demo of strcmp() and strstr()
 */
int main() {
    char str1[] = "Hello, world", str2[] = "world";

    if (strcmp(str1, str2) == 0)
        printf("The strings \"%s\" and \"%s\" are the same\n", str1, str2);
    else
        printf("The strings \"%s\" and \"%s\" are different\n", str1, str2);

    if (strstr(str1, str2) != 0)
        printf("String: \"%s\" DOES contain the string \"%s\"\n",
              str1, str2);
    else
        printf("String: \"%s\" does NOT contain the string \"%s\"\n",
              str1, str2);

    return 0;
}
```

The output is:

```
The strings "hello, world" and "world" are different
String: "hello, world" DOES contain the string "world"
```

Other useful features

String concatenation

Sometimes we wish to use a very long character string that wraps over the side of the page. C solves this for us by the rule that if two strings (not characters!) are separated by white-space they are joined together. Since such strings tend to have new-lines in them, this is a convenient place to break them although it is not compulsory. For example:

```
printf("Menu\n\n"
       "1. Hot dog\n"
       "2. Burger\n"
       "2. Cheeseburger\n"
       "4. Veggie-burger\n"
       "5. Double espresso\n"
       "6. Coffee with milk and stuff\n\n");
```

The above seven lines are interpreted as one huge string. It should be noted that C does not insert spaces when joining strings together, if we want spaces we must do that for ourselves within the individual strings. Also, there are no commas between the strings, if there were they would be treated as seven separate strings, not one large one.

"Printing" into strings with `snprintf()`

The `snprintf()` functions acts like `printf()` except that it takes two additional arguments before the format: the name of a character array for the output to go into and the maximum number of bytes to be written to it (which is usually just the length of the array). It "prints" the output into the character array rather than to the screen. It is defined inside `stdio.h`, just like `printf()`.

There is also a function `sprintf()` (no 'n') that omits the protection of the maximum length. We do not recommend you use it.

The following example demonstrates `snprintf()` protecting us against trying to write some text into a buffer that is not large enough.

```
#include <stdio.h>

#define N 8

/*
 * Demonstrate snprintf protecting against a buffer overflow.
 * We have "accidentally" made the buffer too short for the text.
 */
int main() {
    char buffer[N];
    int i;

    snprintf(buffer, N, "Hello, world\n");

    /* Print out the individual bytes for information */
    for(i = 0; i < N -1; ++i)
        printf("Byte %d: %d\t'%c'\n", i, buffer[i], buffer[i]);

    printf("Final byte: %d\n", buffer[N-1]);

    printf("%s", buffer);

    return 0;
}
```

The output is:

```
Byte 0: 72      'H'
Byte 1: 101     'e'
Byte 2: 108     'l'
Byte 3: 108     'l'
Byte 4: 111     'o'
Byte 5: 44      ','
Byte 6: 32      ' '
Final byte: 0
```

Notice that unlike `strncpy()`, `snprintf()` does the right thing with the final zero byte.

Reading from strings with `sscanf()`

Most of the time we want to read in data from the keyboard or from a file stored on the computer. Occasionally, however, we may have a text string that contains the data we want to "read in".

The `sscanf()` function is exactly like `fscanf()` except that the first argument is a string or character array which is used as the source of the data, rather than an external file. It can be thought of as the opposite to `snprintf()` and is also defined inside `stdio.h`.

Example

```
char mystring[] = "12 34";
int j, k;

sscanf(mystring, "%d %d", &j, &k);
```

The variables `j` and `k` take their values from the character array `mystring[]` and hence have their values set to 12 and 34 respectively. There is no input from any file or the keyboard and `mystring[]` is not altered in any way.
