

Strings as arrays, as pointers, and string.h

1. Strings as arrays:

In C, the abstract idea of a string is implemented with just an array of characters. For example, here is a string:

```
char label[] = "Single";
```

What this array looks like in memory is the following:

```
-----
| S | i | n | g | l | e | \0 |
-----
```

where the beginning of the array is at some location in computer memory, for example, location 1000.

Note: Don't forget that one character is needed to store the *nul character* ($\backslash 0$), which indicates the end of the string.

A character array can have more characters than the *abstract string* held in it, as below:

```
char label[10] = "Single";
```

giving an array that looks like:

```
-----
| S | i | n | g | l | e | \0 |   |   |   |
-----
```

(where 3 array elements are currently unused).

Since these strings are really just arrays, we can access each character in the array using subscript notation, as in:

```
printf("Third char is: %c\n", label[2]);
```

which prints out the third character, **n**.

A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

```
char label[] = "Single"; /* 7 characters */
char label[10] = "Single";
```

Thus, you must specify the maximum number of characters you will ever need to store in an array. This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

2. Strings as pointers:

Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.

Since we are talking about *strings*, which are made up of *characters*, we'll be using *pointers to characters*, or rather, `char *`'s.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a `char *` to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a *character pointer* to keep track of a string.

```
char label[] = "Single";
char label2[10] = "Married";
char *labelPtr;

labelPtr = label;
```

We would have something like the following in memory (e.g., supposing that the array `label` started at memory address `2000`, etc.):

```
label @2000
-----
| S | i | n | g | l | e | \0 |
-----

label2 @3000
-----
| M | a | r | r | i | e | d | \0 | | |
-----

labelPtr @4000
-----
| 2000 |
-----
```

Note: Since we assigned the pointer the address of an *array of characters*, the pointer must be a *character pointer*--the types must match.

Also, to assign the address of an array to a pointer, we do not use the *address-of* (`&`) operator since the name of an array (like `label`) behaves like the address of that array in this context. That's also why you don't use an ampersand when you pass a string variable to `scanf()`, e.g,

```
int id;
char name[30];

scanf("%d%s", &id, name);
```

Now, we can use `labelPtr` just like the array name `label`. So, we could access the third character in the string with:

```
printf("Third char is: %c\n", labelPtr[2]);
```

It's important to remember that the only reason the pointer `labelPtr` allows us to access the `label` array is because we made `labelPtr` point to it. Suppose, we do the following:

```
labelPtr = label2;
```

Now, no longer does the pointer `labelPtr` refer to `label`, but now to `label2` as follows:

```
label2 @3000
-----
| M | a | r | r | i | e | d | \0 |   |   |
-----

labelPtr @4000
-----
| 3000 |
-----
```

So, now when we subscript using `labelPtr`, we are referring to characters in `label2`. The following:

```
printf("Third char is: %c\n", labelPtr[2]);
```

prints out `r`, the third character in the `label2` array.

3. Passing strings:

Just as we can pass other kinds of arrays to functions, we can do so with strings.

Below is the definition of a function that prints a label and a call to that function:

```
void PrintLabel(char the_label[])
{
    printf("Label: %s\n", the_label);
}

...

int main(void)
{
    char label[] = "Single";
    ...
    PrintLabel(label);
    ...
}
```

Since `label` is a character array, and the function `PrintLabel()` expects a character array, the above makes sense.

However, if we have a pointer to the character array `label`, as in:

```
char *labelPtr = label;
```

then we can also pass the pointer to the function, as in:

```
PrintLabel(labelPtr);
```

The results are the same. *Why??*

Answer: When we declare an array as the parameter to a function, we really just get a pointer. Plus, arrays are always automatically passed by reference (e.g., a pointer is passed).

So, `PrintLabel()` could have been written in two ways:

```
void PrintLabel(char the_label[])
{
    printf("Label: %s\n", the_label);
}
```

OR

```
void PrintLabel(char *the_label)
{
    printf("Label: %s\n", the_label);
}
```

There is no difference because in both cases the parameter is really a *pointer*.

Note: In C, there is a difference in the use of brackets (`[]`) when declaring a global, static or local array variable *versus* using this array notation for the parameter of a function.

With a parameter to a function, you always get a *pointer* even if you use array notation. This is true for **all** types of arrays.

4. Dynamically-allocated string:

Since sometimes you do not know how big a string is until run-time, you may have to resort to dynamic allocation.

The following is an example of dynamically-allocating space for a string at run-time:

```
#include <stdlib.h> /* for malloc/free */

...

void SomeFunc(int length)
{
    char *str;

    /* Don't forget extra char for nul character. */
    str = (char *)malloc(sizeof(char) * (length+1));

    ...
}
```

Basically, we've just asked `malloc()` (the allocation function) to give us back enough space for a string of the desired size. `malloc()` takes the number of bytes needed as its parameter. Above, we need the size of one character times the number of characters we want (don't forget the extra +1 for the *nul character*).

We keep track of the dynamically-allocated array with a pointer and can use that pointer as we used pointers to statically-allocated arrays above (i.e., how we access individual characters, pass

the string to a function, etc. are the same).

Now, how do we get a string value into this newly-allocated array?

5. string.h library:

Recall that strings are stored as arrays (allocated either statically or dynamically). Furthermore, the only way to change the contents of an array in C is to make changes to each element in the array.

In other words, we can't do the following:

```
label = "new value"; /* No! */
label = anotherLabel; /* Wrong! */
```

(where `anotherLabel` is a string variable).

Aside: We could do that if `label` was a character pointer (instead of an array); however, what would be happening is the pointer would be taking on the address of a different string, which is not the same as changing the contents of an array.

It would be annoying to have to do something like:

```
char name[10];

name[0] = 'R';
name[1] = 'o';
name[2] = 'b';
name[3] = '\0';
```

or to write *loops* all the time to do common string operations... Plus, we'd probably forget the *nul character* half the time.

The C library `string.h` has several common functions for dealing with strings. The following four are the most useful ones that we'll discuss:

- `strlen(str)`

Returns the number of characters in the string, not including the *nul character*.

- `strcmp(str1, str2)`

This function takes two strings and compares them. If the strings are equal, it returns **0**. If the first is greater than the 2nd, then it returns *some value greater than 0*. If the first is less than the 2nd, then it returns *some value less than 0*.

You might use this function as in:

```
#include <string.h>

char str1[] = "garden";

if (strcmp(str1, "apple") == 0)
    printf("Equal\n");
else
```

```
printf("Not equal\n");
```

OR

```
if (strcmp(str1, "eden") > 0)
    printf("'s' comes after 'eden'\n", str1);
```

The ordering for strings is lexical order based on the ASCII value of characters. Remember that the ASCII value of 'A' and 'a' (i.e., upper/lowercase) are not the same.

An easy way to remember how to use `strcmp()` to compare 2 strings (let's say *a* and *b*) is to use the following mnemonics:

Want...	Use...
<code>a == b</code>	<code>strcmp(a, b) == 0</code>
<code>a < b</code>	<code>strcmp(a, b) < 0</code>
<code>a >= b</code>	<code>strcmp(a, b) >= 0</code>
...	...

- `strcpy(dest, source)`

Copies the contents of *source* into *dest*, as in:

```
#include <string.h>
char str1[10] = "initvalue";
strcpy(str1, "second");
```

Now, the string `str1` contains the following:

```
-----
| s | e | c | o | n | d | \0 | u | e | \0 |
-----
```

and the word "initvalue" has been overwritten. Note that it is the first *nul character* (`\0`) that determines the end of the string.

When using `strcpy()`, make sure the *destination* is big enough to hold the new string.

Aside: An easy way to remember that the destination comes first is because the order is the same as for assignment, e.g:

```
dest = source
```

Also, `strcpy()` returns the destination string, but that return value is often ignored.

- `strcat(dest, source)`

Copies the contents of *source* onto **the end of** *dest*, as in:

```
#include <string.h>

char str2[10] = "first";

strcat(str2, " one");
```

Now, the string `str2` contains the following:

```
-----
| f | i | r | s | t |   | o | n | e | \0 |
-----
```

When using `strcat()`, make sure the *destination* is big enough to hold the extra characters.

Aside: Function `strcat()` also returns the destination string, but that return value is often ignored.

BU CAS CS - Strings as arrays, as pointers, and string.h
Copyright © 1993-2000 by Robert I. Pitts <rip at bu dot edu>. All Rights Reserved.