

A Rudimentary Intro to C programming

Wayne Goddard

School of Computing, Clemson University, 2008

Part 4: Strings and Pointers

18	Strings	D1
19	String Functions	D3
20	Pointers	D5
21	Strings and Pointers	D7
22	More on Pointers	D10

Strings

18.1 Strings are Null-terminated Arrays

We have used strings with double quotes—these are constant strings. But what about testing and changing strings? In C, a string is a null-terminated sequence of characters; that is, there is a special character—written `'\0'`—after the last normal character.

The standard approach for a user-created string is to store it inside a character array. In particular, this means that the array must have size at least 1 more than the length of the string. For example, a string like `"happy"`, which has length 5, is stored in a `char` array of size at least 6 as:

0	1	2	3	4	5	6 onwards
h	a	p	p	y	\0	irrelevant

Constant strings, like the ones we provide to `printf`, automatically have the null character added to them. But any user-created string must have `'\0'` explicitly added.

To print a string with `printf` or read a string with `scanf`, use `%s`. Note that `scanf` is passed just the name of the `char` array (we'll see why later)—no ampersand. Also, `scanf` ignores whitespace before the string and treats a whitespace as the end of string. So, if you want to read a string that might have a space in it, you need to read in one char at a time, using `getchar()`.

18.2 Example Program: `stringRead.c`

```
// stringRead.c
// adapted from Jamsa
// read a string from user and echo it
#include <stdio.h>
const char EOL = '\n';
const int SIZE = 100;

int main(void) {
    char stringA[SIZE], stringB[SIZE];
    int index;
    char letter;

    // read a string using getchar
```

```

printf("Enter string A: ");
index = 0;
letter = getchar();
while( letter!= EOL ) {
    stringA[index] = letter;
    index++;
    letter = getchar();
}
stringA[index] = '\0';

// read a string using scanf
printf("Enter string B: ");
scanf("%s", stringB);

printf("The first string was: %s\n", stringA);
printf("The second string was: %s\n", stringB);
return 0;
}

```

has sample output:

```

Enter string A: happy days
Enter string B: happy days
The first string was: happy days
The second string was: happy

```

Practice Adapt the above program to read a sentence, terminated by a period.

String Functions

You can create your functions to do many things with strings. For standard tasks, there are function in the `string` library.

19.1 Creating Your Own String Function

Almost all string functions have a main loop that iterates until the end of the string is detected. For example, here is code to compute the length of a string:

```
int strlen(char s[]) {
    int x = 0;
    while (s[x] != '\0')
        x=x+1;
    return x;
}
```

It returns 5 when called by

```
char test[] = "happy";
strlen(test);
```

Or suppose you wanted to convert a string to all capitals:

```
void toUpperCase(char s[]) {
    int x;
    for( x=0; s[x] != '\0'; x++ ) {
        if( s[x]>='a' && s[x]<='z')
            s[x] += 'A' - 'a';
    }
}
```

19.2 String Library

Further string functions are available in the `string` library. But beware! If the array you are copying to is not big enough, then crash: the null-character terminator gets lost. These functions include:

- `strcpy(dest,src)` copies one string into another
- `strcat(dest,src)` appends one string to another

- `strlen(s)` returns the string length
- `strcmp(first,second)` compares the strings alphabetically: it returns a negative value if the first comes before the second, 0 if the two strings are the same, and a positive value if the first comes after the second.

The only one you have to know is `strlen`.

19.3 Example Program: `stringLibrary.c`

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

int main(void) {

    char alpha[] = "alpha";
    char beta[] = "beta";
    char final[100];

    strcpy(final, alpha);
    if( strcmp(final,alpha)==0 ) // true
        printf("%s and %s are the same\n", final, alpha );

    strcat(final,beta); // final is now "alphabetabeta"
    if( strcmp(final,alpha)>0 ) // true
        printf("%s comes after %s\n", final, alpha );

    return 0;
}
```

Pointers

20.1 Computer Memory

We have thus far ignored the details of how variables are stored in memory. Indeed, in some high-level languages this ignorance is encouraged. But in C one is encouraged to know and exploit how variables are stored.

Memory is arranged as a collection of cells numbered from 0 up to how ever many is available on the machine. A single variable is stored in a cell, or in a block of cells, depending on its size. We shall see that a cell is big enough to store a single character; but a string is stored in a block of cells. Indeed, any array is stored as a block of cells.

So what happens when we define a variable: say `int A`? The compiler and operating system together choose a memory cell to store `A`—you can think of a name tag stuck on the cell door. But this cell also has a number—what we call the *address* of `A`. We have already seen that `&A` will give us this address. But other than passing that information to `scanf`, we have not used this address.

20.2 Pointers, Addresses and Dereferences

The language C has a data-type called a *pointer*. This is a variable that stores the address of another variable. Pointers use the `*` notation. In fact pointer syntax uses stars in two *different* meanings. We declare that a variable is a pointer by prefacing it with a star:

```
int *p;
```

Note that while an address is really just a number, a pointer holds the address of a variable of a specific type. That is, the pointer `p` stores the address of a variable of type `int`. (Well, actually, at the start `p` contains garbage, until we initialize it.) We say it points to a specific cell in memory.

We initialize a pointer by setting it to the address of a variable. This can be done in *two* ways: either set it to the address of some variable using the `&` notation, or set it to another pointer. Here is the former:

```
int *p = &A;
```

Now, we need to be able to work with the variable that `p` points to. This value is accessed by `*p` (sometimes called *dereferencing*).

The following code creates a variable `x` and a pointer `p` to it. Once that is set up, changing the value of `*p` changes the value of `x` and vice versa.

```
int x = 2;
int *p = &x;
printf("%d",*p); // prints 2
*p = 11;
printf("%d",x); // prints 11
x = -5;
printf("%d",*p); // prints -5 because p still points to x
```

20.3 Pointer Arithmetic and Arrays

Incrementing a pointer moves it to the next “cell” in memory. (The actual change in the address depends on the size of the variable that the pointer points to.) This pointer arithmetic is useful in array manipulation. The pointer can be initialized with the name of the array—because the name of the array is equivalent to the address of the first cell. Indeed, the name of the array is a pointer.

The following code prints out the array:

```
int B[] = {1,4,9,16};
int *p = B; // initialize p to point to start of array B
for(int i=0; i<4; i++) {
    printf("%d", *p);
    p++;
}
```

It is good practice to set unused pointers to NULL (a special constant defined in C to be equal to 0), to emphasize that we know it’s not yet or no longer usable.

Strings and Pointers

21.1 Strings via Pointers

We have seen that in C a string is stored as an array of `chars`. It turns out that any block of `chars` can be interpreted as a string. This block can be accessed via the array name, or more generally, via a pointer.

In general, a string function can take as its argument either an array name or a pointer to the first char in the string. Here is another implementation of the function to calculate the length of a string, using pointer arithmetic.

```
int strlen(char *s) {
    int len=0;
    while (*s != '\0') {
        s++; len++;
    }
    return len;
}
```

We can use pointer arithmetic to achieve the same effect:

```
int strlen(char *s) {
    char *t = s;
    while (*t != '\0')
        t++;
    return (t-s);
}
```

Here is a further implementation of `strlen` that uses the fact that the `'\0'` character is equal to zero when used in a calculation. That is, the boolean condition `*s` is true if `*s` is nonzero, meaning `s` points to anything except the `'\0'` character. It also uses the comma construct, which for example allows two statements to be included in the update portion of a `for` loop.

```
int strlen(char *s) {
    int x=0;
    for ( ; *s; s++, x++ ) // notice the comma
        ; // empty body
    return x;
}
```

So, most of the time when you see `char *P` written, the pointer `P` is going to point to (the first char in) some string. But sometimes it really points only to a single char. Only later code (or comments!) can clarify which.

21.2 Example Program: `strstr.c`

The following program contains a function that searches for the first occurrence of one string (called `needle`) inside another (called `haystack`).

```
// strstr.c - adapted from nkraft
// Find first occurrence of substring needle in string haystack.
#include <stdio.h>

// The function returns pointer to beginning of substring,
// or NULL if substring is not found.
char *strstr ( char *haystack, char *needle ) {
    char *start;
    for(start = haystack; *start != '\0'; start++ ) {
        char *p = needle;
        char *q = start;
        while ( *p != '\0' && *q != '\0' && *p == *q ) {
            p++;
            q++;
        }
        if( *p == '\0' )
            return start;    // reached end of needle without mismatch
    }
    return NULL;
}

int main ( void ) {
    char *one = "concatenate";
    char *two = "cat";

    char *ans = strstr( one, two );
    if ( ans == NULL )
        printf( "needle not found\n" );
    else
        printf( "needle starts in position %d of haystack\n", ans-one );
}
```

```
    return 0;
}
```

21.3 Changing Strings

One can also write code that alters a string. For example, what does the following code do to a string that was defined by `char *beth`?

```
while( *beth==' ' )
    beth++;
```

It removes any leading spaces.

But note that this assumes that `beth` is stored in an array. If one writes a declaration as `char *beth = " happy"`, the string “happy” is a constant and not alterable (you’ll get a segmentation fault).

More on Pointers

22.1 Pass by Value

In C, all function arguments are passed by value. That is, the function receives a copy of the value, and cannot change the original. So while the contents of an array that is passed to a function can change, ints passed to a function cannot change. And yet `scanf` does its job. . .

The solution is that you can pass the function the *address* of a variable. For example, a function to swap the values of two numbers uses pointers:

```
// swaps the values pointed to by a and b
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

The calling code might look something like:

```
int x,y;
// x and y initialized
swap( &x, &y );
```

An array is automatically passed as an address, equivalent to a *pointer* to the base type.

22.2 Dynamic Memory Allocation

One can create an array on the fly: this uses dynamic memory allocation. This is especially important for arrays (and more advanced structures) that exist only temporarily.

The function `calloc` is contained in `stdlib.h`. It reserves/allocates and initializes a block of memory. The function takes two `int` arguments: the first argument is the number of elements to be stored and the second argument is the size in bytes (think “cells”) of the element that is to be stored (which is obtained using the `sizeof`). The `calloc` function returns the address of the beginning of the block.

```
int *intPtr;
intPtr = calloc( 1000, sizeof(int));
*intPtr = 10;
```

will create an array with 1000 elements and set the first value to 10 (the rest being zero).

Memory that is allocated with `calloc` must be released afterwards using the `free` command: `free(intPtr);` This allows memory to be recycled.