**Array & Pointer**

Let us recollect why we require pointers and how arrays and pointers are always together.

a) Pointers are required to mimic parameter passing by reference. If we have to change a variable by calling a function, we have to pass pointer to the variable. The corresponding parameter will be of pointer type.

b) Array degenerates to a constant pointer at runtime. So, we can assign an array to a pointer. When we pass an array as argument to a function, the corresponding parameter is a pointer. Parameter can never be an array.

Do not ever worry about the value of a pointer.

Understand the following expression clearly.

a)

int a[10]; int \*p = a;

int i = <some value between 0 and 9>;

a[i], \*(a + i), p[i], \*(p + i) are all equivalent.

a + i, &a[i], &p[i], p + i are all equivalent.

b)

int a[] = {11, 22, 33, 44};

int \*p = a + 2;

printf(“%d”, (\*p)++);

The value of the expression is the old value of \*p which is33. Then \*p is incremented. So, a[2] becomes 34.

c)

int a[] = {11, 22, 33, 44};

int \*p = a + 2;

printf(“%d”, \*p++);

\* and ++ are both unary. The association is right to left. The variable p is incremented. The value of p++ is the old value of p as it is post increment operator. That old value of p stored in a temporary is dereferenced.

Display is 33. p will point to a[3].

Let us have a look at the following code.

int a[] = {11, 22, 33, 44, 55};

// a : array at compile time

int n = sizeof(a) / sizeof(\*a); // sizeof(a) / sizeof(int)

printf("n : %d\n", n);

sizeof is a compile time operator. sizeof of an array returns the number of bytes occupied by the array. sizeof array / sizeof component gives us the number of elements in the array. But this can be applied at the place where the array is declared. We cannot use it on parameters as the parameter is always a pointer when the corresponding argument is an array – the parameter does not know that it refers to an array. At run time, array does not keep track of its size.

The group of program files (1\_array.c, 1\_array,h, 1\_client.h, 1.mk) shows how to pass array as an argument and display it.

**Strings:**

The language ‘C’ distinguishes between character and string. ‘C’ does not have any basic type called string even though there are functions to play with strings.

‘a’ : character

“a” : string

A string is like an array of characters. Unlike normal arrays, strings are terminated by a character called NULL character whose ASCII value is 0. It is same as ‘\0’. So character ‘a’ occupies 1 byte where as string “a” occupied 2 bytes - ‘a’ ‘\0’.

Let us look at some examples from the file 2\_c\_str.c.

The string “pes univ” occupies 9 characters. One char in the array a is not used.

char a[10] = "pes univ";

The compiler counts and allocates an array of 7 characters (and not 6!) for array b.

char b[] = "python";

char c[] = { 'p', 'y', 't', 'h', 'o', 'n', '\0' };

The above two statements are equivalent. Initializing a c string as in the case of array b is a short cut available only for c strings.

We can read and write c strings using %s format in scanf and printf. scanf with %s option will introduce NULL character at the end of the string. Printf will display the characters until NULL character is encountered.

printf("str a : %s\n", a);

printf("str b : %s\n", b);

printf("str c : %s\n", c);

Effectively the string d is “cat”.

char d[] = { 'c', 'a', 't', '\0', 't', 'l', 'e', '\0' };

printf("d : %s\n", d);

for(int i = 0; i < 8; ++i)

{

putchar(d[i]);

}

putchar('\n');

If the string is hand crafted, it is our responsibility to end the string by NULL character. All builtin string functions expect this sentinel value ‘\0’ failing which the results are undefined.

Let us write three functions to get a feel how these string functions operate.

Check the files : 2\_c\_str.c 2\_str\_fn.h 2\_str.fn.c

**Length of a string.**

The client code:

char e[10] = "india";

printf("length : %d\n", mystrlen(e));

The implementation:

version 1:

In this function, we access the elements of the array through an index i. We stop when we encounter the NULL character. The value of i is the length of the string. You may want to think why is it not that plus 1.

int mystrlen(char s[])

{

// version 1

int i = 0;

while(s[i] != '\0')

{

++i;

}

return i;

}

version 2: This uses the pointer to access the characters in the string.

As the expression of while, we may use any of these three expressions and all are equivalent. Would changing the pointer here affect the argument?

int mystrlen(char s[])

{

// version 2

int i = 0;

// while(\*s++ != '\0')

// while(\*s++ != 0)

while(\*s++ )

{

++i;

}

return i;

}

version 3:

This is recursive. The length of a string whose first character is NULL is zero. Otherwise add 1 to the length of the string starting from next position.

You may want to think the difference between mystrlen(s++) and mystrlen(s + 1).

int mystrlen(char s[])

{

// version 3:

if(\*s == '\0')

{

return 0;

}

else

{

//return 1 + mystrlen(s++); // infinite recursion

return 1 + mystrlen(s + 1);

}

}

**copy string:**

we are trying to copy the string f to string g. The string g should have enough space to hold the string f. Otherwise, it is an undefined behaviour.

The client code:

mystrcpy(g, f);

The implementation:

version 1:

Copy the characters from src to dst until the element in the src is a NULL character. We should not forget to copy the NULL character to dst after the loop. Otherwise all functions called on dst would be in trouble.

Can we say : dst[i++] = src[i]; instead of 2 lines in the loop?

void mystrcpy(char \*dst, char \*src)

{

int i = 0;

while(src[i] != '\0')

{

dst[i] = src[i];

++i;

}

dst[i] = '\0';

}

version 2:

This is an amazing code. \*src++ returns the char to which src was pointing before incrementing src. \*dst++ returns the location to which dst was pointing before it is incremented. We copy the char returned by \*src++ to \*dst++. If the assigned char is not false(not NULL), we keep doing nothing in the loop.

Nothing to do after the loop as well.

void mystrcpy(char \*dst, char \*src)

{

while(\*dst++ = \*src++)

;

}

**compare two strings:**

The client code:

printf("compare : %d\n", mystrcmp("amar", "amar")); // expect to get 0

printf("compare : %d\n", mystrcmp("amar", "akbar")); // expect +ve value

printf("compare : %d\n", mystrcmp("amar", "anthony")); // expect -ve value

The implementation:

int mystrcmp(char \*s1, char \*s2)

{

while(\*s1 && \*s2 && \*s1 == \*s2)

{

++s1; ++s2;

}

return \*s1 - \*s2;

}

The concept is like this. If the two strings matched till the end including the NULL character – take the difference of these NULL characters – return 0 as the result.

Keep comparing the corresponding characters from left to right until a mismatch. Subtract the first character from the second. If the strings are in order based on ASCII, the difference will be negative – otherwise positive.

Can we say s1 – s2 instead of \*s1 - \*s2?

You may want to think.