Arrays and Pointers: Strings

Strings are one-dimensional arrays of type char. However, a string is stored slightly differently compared to an array of `int` or `double`. Further, we can use many functions (usually defined in `string.h`) that can be used exclusively for strings. Hence, this lecture focuses exclusively on strings.

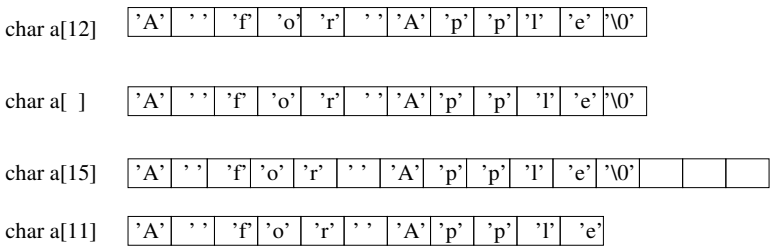
By convention, a string in C is terminated by the end-of-string sentinel `'\0'`, or null character. The size of the string must be such that it includes the end-of-the-string null character. If we consider the string `"A for Apple"`, we see that it requires at least 12 spaces (9 letters, two blank spaces and one space for the null character) to store it. To declare it, we may declare any of the following (see the figure below):

i. `char a[12]={'A',' ','f','o','r',' ','A','p','p','l','e','\0'};`
   where we have declared the array size to the exact value required.

ii. `char a[]={'A',' ','f','o','r',' ','A','p','p','l','e','\0'};`
   where the size is taken to be 12, and hence it is equivalent to the previous declaration.

iii. `char a[15]={'A',' ','f','o','r',' ','A','p','p','l','e','\0'};`
   where the last three spaces are not used.

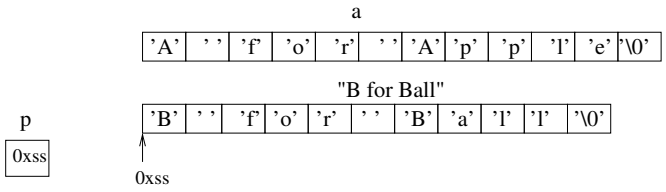
Note that instead of writing a single quote around each character, the above declarations are typically implemented using `char a[12]="A for Apple"`, or `char a[]="A for Apple"`, or `char a[15]="A for Apple"`.



On the other hand, if we define `char a[11]="A for Apple"` (last panel in figure), then though it is an array of characters, it is not well-behaved with string-functions since the end-of-string null character is missing here.

Now consider the following code segment.

```
char a[]="A for Apple";
char *p="B for Ball";
printf("%s\n",a); //A for Appple   is printed
printf("%s\n",p); //B for Ball     is printed
printf("%s\n",a+2); //for Apple    is printed
printf("%s\n",p+2); //for Ball     is printed
printf("%c\n","C for Cat"[2]); //f is printed
p="C for Cat";
```



Note that the two declarations, though they look similar, have different interpretations. The first declaration defines `a` to be an array of characters of size 12, and it is initialized as shown in the first panel of the figure shown above. Since the array name `a` is, by definition, the address of the first element, the argument of the first `printf()` statement is the address of the first element. Similarly, `a+2` is the address of the third element of the array `a`. When a pointer to char is printed in the format of a string, the pointed-at character and each successive character are printed until the end-of-string null character is reached. Hence, `A for Apple` and `for Apple` are printed for the first and third `printf()` statements, respectively. Now we look at the second declaration. Here `p` is a pointer to char, and it has been assigned the address of the string constant `"B for Ball"`. Note that for the string constant `"B for Ball"`, the name of the array is `"B for Ball"` and it is initialized as shown in the 2nd panel of the figure above. Since the char pointer `p` has the address of first element of the string constant, the output of the `printf()` statements of the code segment involving `p` is obvious. Since string constant `"C for Cat"` is the name of the array itself, the expression `printf("%c\n","C for Cat"[2])` prints the 3rd element ('e') of the array. The last expression `p="C for Cat"` assigns the address of the first location of the string constant `"C for Cat"` to `p`.

Note that though we write `p="C for Cat"` in the above code segment, writing `a="C for Cat"` is not allowed. This is because `a` is a constant array whose base address is fixed. Also, observe that `'s'` and `"s"` are not the same. The first is a character constant consisting of a single character `'s'`, but the latter is a string constant consisting of two characters `'s'` and `'\0'`.

String-Handling Functions: These functions are typically defined in `string.h`. Many of these functions have argument `const char *`, where a `const char *` is a pointer to a character that cannot be modified through that pointer. It is commonly used for constant character arrays, ensuring data integrity by preventing accidental modifications. While the content is constant, the pointer itself can be updated. Some widely used string-handling functions are given below.

i. `char *strcat(char *dest, const char *src)`: This function is used to concatenate (join) two strings. It appends the source string (`src`) to the destination string (`dest`). The function returns a pointer to the destination string. The programmer must ensure that the destination string has enough space to hold the result. The return value of `strcat()` is generally not assigned because it returns the exact same pointer to the destination string that was passed as the first argument. Since the caller already holds this pointer, assigning the return value is redundant.

ii. `int strcmp(const char *str1, const char *str2)`: This function is used to com-

pare two strings terminated by a null character lexicographically (alphabetically). An integer is returned that is less than, equal to, or greater than zero, depending on whether `str1` is lexicographically less than, equal to, greater than `str2`.

iii. `char *strcpy(char *dest, const char *src)`: This function is used to copy the contents of the source string (`src`) to the destination string (`dest`). The characters in the string `src` are copied into `dest` until `'\0'` is moved. Whatever exists in `dest` is overwritten. It is assumed that `dest` has enough space to hold the result. The pointer to `dest` is returned. The return value of `strcpy()` is again generally not assigned because it returns the exact same pointer to the destination string that was passed as the first argument.

iv. `size_t strlen(const char *src)`: This function is used to calculate the length of a source string (`src`), excluding the null character `'\0'`. It returns a value of type size_t (an unsigned long integer).

Note that we can easily write these functions ourselves. Below, we write a function `astrcpy()`, whose job is the same as the `strcpy()` function.

```
#include <stdio.h>
char *astrcpy(char *,const char *);
int main() {
    char src[30] = "the doctors away!";
    char dest[100]="An Apple a day keeps ";
    printf("Src. string before astrcpy: %s\n", src);  // Print the source string
    printf("Dest. string before astrcpy: %s\n", dest); // print the destination string
    astrcpy(dest, src);  // Call the user defined copy function
    printf("Dest. string after astrcpy: %s\n", dest); // print destination  string
    return 0;
}
char *astrcpy(char *dest, const char *src)
{
 char *p; // to move along the dest string
 int i=0;
 p=dest;
 while(src[i]!='\0')
 {
  p[i]=src[i];
  i++;
 }
 p[i]='\0'; // insert '\0' at the end
 return dest;
```

```
}
```

Listing 54: C Program "cpystr.c"

Below is the output.

```
Src. string before astrcpy: the doctors away!
Dest. string before astrcpy: An Apple a day keeps
Dest. string after astrcpy: the doctors away!
```

Finally, we write a program in which we illustrate the string handling functions introduced above.

```
#include <stdio.h>
#include <string.h>
int main() {
    char src[30] = "the doctors away!";
    char dest[100]="An Apple a day keeps ";
    printf("Src. string: %s\n", src);  // Print the source string
    printf("Dest. string: %s\n", dest); // print the destination string
    printf("strcmp of src. and dest.: %d\n", strcmp(src,dest)); // call the function s
    printf("The length of dest. string: %zu\n",strlen(dest)); // call the function str
    strcat(dest, src);  // Call the function strcat()
    printf("Dest. string after strcat: %s\n", dest); // print concatenate string
    printf("Src. string  after strcat: %s\n", src); // print src. string
    strcpy(dest,src);  // call the function strcpy()
    printf("Dest. string after strcpy: %s\n", dest); // print concatenate string
    printf("Src. string  after strcpy: %s\n", src); // print src. string
    return 0;
}
```

Listing 55: C Program "strhnd.c"

Below is the output:

```
Src. string: the doctors away!
Dest. string: An Apple a day keeps
strcmp of src. and dest.: 51
The length of dest. string: 21
Dest. string after strcat: An Apple a day keeps the doctors away!
Src. string  after strcat: the doctors away!
Dest. string after strcpy: the doctors away!
Src. string  after strcpy: the doctors away!
```

When we use `strcmp()` between `src="the doctors away!"` and `dest="An Apple a day keeps "`, we start comparison from the first character. Now `'t'` (in `src`) has integer value `116` and `'A'` (in `dest`) has integer value `65`. Thus `116-65=51` is the output. Next, we compute the length of the `dest` string using `strlen()`, which returns an `int` of type `size_t` (unsigned long), which can be printed using `%zu`. Note that `dest` string has 16 letters and 5 blank spaces (note a blank at the end, i.e., before `'\0'`). Thus, the output is `21`. After the call to `strcat()`, the `src` string is appended at the end of the `dest` string. Thus, before the call to `strcpy()`, the string `dest="An Apple a day keeps the doctors away!"` and the string `src="the doctors away!"`. After the call to `strcpy()`, the `dest` string is overwritten with the `src` string.

Return to Lecture Topics