C strings Strou. (5.2-5.3)

C-string: null terminated array of char

char greet[5] = { 72, 73, 76, 65, 0 };
char greet[ ] = { 'H', 'I', 'L', 'A', '\0' };//"AUTO COUNT"
char greet[ ] = "HILA"; // Initializer shorthand
.data

greet: .byte 72
    .byte 73
    .byte 76
    .byte 65
    .byte 0

greet: .ascii "HILA"
**String Literal:** "This is a string" (The null char is added.)

Except when it is used to initialize a char array, a string literal is
“array of the appropriate number of const characters”

Since an array name signifies a **pointer**, a string literal can be
assigned to a **char**

```c
char greet[] = "HILA";
    //IMPORTANT
    //    DIFFERENCE!!
char * pchar = "LUKY";

greet[1] = 'E';    // OK
pchar[1] = 'E';    // BAD!!
cout << "LUKY";
    // might now print LEKY
```

---

**Slide 3**

```c
char greet[] = "HILA";
char * pchar = "LUKY";

pchar = greet;

greet[0] = 'A';
pchar[1] = 'B';
*(greet+2) = 'C';
*(pchar+3) = 'D';
cout << greet << pchar;
    // prints ABCDABCD
pchar = pchar + 1;
cout<< pchar << ' '<<greet;
    // prints BCD ABCD
```
Correspondence summary:

\&a[0] : Synonymous with a
a[0] : Synonymous with *a
\&a[i] : Synonymous with a+i
a[i] : Synonymous with *(a+i)

Equivalent statements:

\begin{align*}
a[8] &= '!' ; & *(a+8) &= '!' ; \\
px &= a + 8 ; & px &= a ; \\
px &= + = 8 ; &
\end{align*}

Sample program segment:

\begin{verbatim}
char a[10] ; char *px = a ;
*(a+8) = '!' ; px += 8 ;
cout << a[8] << " " << *px << endl ;
\end{verbatim}

Output:

! !
Summary: After the code segment

```c
char a[10]; char *px = a;
```

the characters in `a` can be accessed as

```c
a[0], a[1], ..., a[9]
```

or as

```c
*a, *(a+1), ..., *(a+9)
```

or as

```c
*px, *(px+1), ..., *(px+9)
```

---

Program examples (using subscript arithmetic)

**Example 1**: Computing the length of a string. (The length does not include the ‘\0’ char.)

(a) Array version:

```c
size_t strlen (const char *s) {
    int i = 0;
    while (s[i] != '\0')
    {
        // s[0...i] CONTAINS NO NULL CHAR
        i++;
    }
    // s[0...i-1] CONTAINS NO NULLS, s[i]=='\0'
    return (size_t) i;
}
```
Notes:

- The type `size_t` is usually defined to be `unsigned int` or `unsigned long`.

- The `const` qualifier indicates that string `s` is not modified by the function.

- As a function parameter, "char *s" and "char s[]" are equivalent.

- A typecast is used in the `return` statement.

(b) Using pointer arithmetic:

```c
size_t strlen (const char *s) {
    char *p = s;
    while (*p != '\0')
        { // bytes located from s up to p are NON-NULL
            p++;
        }
    // p == address of first NULL byte
    // located at or after s
    return (size_t) (p-s);
}
```
Notes:

- The statement "char *p = s;" initializes the pointer p.
- The loop could also have been written as follows:

  ```c
  while (*p)
      p++;
  ```

**Reason:** The character '\0' has the (ASCII) value zero. So, the loop would end as soon as that character is seen.

- The expression (p-s) gives the length of the string.

**Example 2:** Copying a string. (The function considered here is slightly different from the `strcpy` function in `<string.h>`.)

**Function header:**

```c
void copy_str (char dest[], char src[])  
```

(a) **Array version:** Exercise.
(b) Pointer version:

```c
void copy_str (char dest[], char src[]) {
    while ( *(dest++) = *(src++) )
        // all copied chars are NON-NULL
    ;  // Empty loop body.
    // Last copied character is NULL
}
```

Notes:
- The function uses the fact that an assignment statement is an expression.
- The assignment statement ensures that the '\0' character is also copied.
- Loop terminates just after copying '\0', since the (ASCII) value of that character is zero.

---

C style string handling library:
- **Documentation**: Stroustrup (20.4.1), p.600-601.
  ```
  man string in Lab.
  (p. 325, etc. Deitel & Deitel)
  ```
- Header file: `<string.h>`
- Tests and manipulations on C strings.
- Functions operate on *null terminated* array of char data.
  They expect strings ending with '\0'

Some examples:
```c
char *strcpy (char *s1, const char *s2)
```
- Copies the string `s2` to `s1`. Returns a pointer to `s1`. (The '\0' character is also copied.)
strcpy( s1, s2 ) is **UNSAFE** if you cannot trust s2. Space at s1 might be smaller than length(s2)+1

```c
#include <string.h>
#include <iostream.h>
main(int argc, char *argv[])
{
    char BUFFER[20]; // 20 ought to be enough REALLY???
    if( argc >= 2 )
        strcpy( BUFFER, argv[ 1 ] );
    else      ... ;
    cout << BUFFER << endl; ...; }
$ a.out csi333
csi333
$ a.out 12345678901234567890123456789012
segmentation error (or bus error)
```

---

**Bounded size string copy function:**

```c
char *strncpy (char *s1,
    const char *s2,
    size_t n)
```

- - Copies *at most* n chars from s2 to s1. Returns a pointer to s1.

- - If s2 has n or more chars, then the *first* n chars are copied to s1.

- - If s2 has less than n chars, then all the chars of s2 are copied to s1. The remaining spots are filled with '\0'.

### Example:

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial:</strong></td>
<td>x y z w q \0</td>
<td>a b c \0</td>
</tr>
<tr>
<td>After <code>strncpy(s1, s2, 3):</code></td>
<td>a b c w q \0</td>
<td>a b c \0</td>
</tr>
<tr>
<td>After <code>strncpy(s1, s2, 5):</code></td>
<td>a b c \0 \0 \0</td>
<td>a b c \0</td>
</tr>
</tbody>
</table>