assigns the same pointer to each of the C-strings name[0], name[1], etc. Arrays cannot be assigned this way. To copy one array into another, use strcpy(), or strncpy().

8.2 This copies the C-string s2 into the C-string s1:

```c
char* strcpy(char* s1, const char* s2)
{ char* p; for (p=s1; *s2; )
    *p++ = *s2++;
    *p = '\0';
return s1;
}
```

The pointer p is initialized at the beginning of s1. On each iteration of the for loop, the character *s2 is copied into the character *p, and then both s2 and p are incremented. The loop continues until *s2 is 0 (i.e., the null character '\0'). Then the null character is appended to the C-string s1 by assigning it to *p. (The pointer p was left pointing to the byte after the last byte copied when the loop terminated.) Note that this function does not allocate any new storage. So its first argument s1 should already have been defined to be a character string with the same length as s2.

8.3 This function appends up to n characters from s2 onto the end of s1. It is the same as the strcat() function except that its third argument n limits the number of characters copied:

```c
char* strncat(char* s1, const char* s2, size_t n)
{ char* end; for (end=s1; *end; end++) // find end of s1
    ;
char* p; for (p=s2; *p && p-s2<n; )
    *end++ = *p++;
*end = '\0';
return s1;
}
```

The first for loop finds the end of C-string s1. That is where the characters from C-string s2 are to be appended. The second for loop copies characters from s2 to the locations that follow s1. Notice how the extra condition p-s2<n limits the number of characters copied to n: the expression p-s2 equals the number of characters copied because it is the difference between p (which points to the next character to be copied) and s2 (which points to the beginning of the C-string). Note that this function does not allocate any new storage. It requires that C-string s1 have at least k more bytes allocated, where k is the smaller of n and the length of C-string s2.

8.4 This requires testing the last letter and the second from last letter of the word to be pluralized. We use pointers p and q to access these letters.

```c
void pluralize(char* s)
{ int len = strlen(s);
    char* p = s + len - 1;    // last letter
    char* q = s + len - 2;    // last 2 letters
    if (*p == 'h' && (*q == 'c' || *q == 's')) strcat(p, "es");
    else if (*p == 's') strcat(p, "es");
    else if (*p == 'y')
        if (isvowel(*q)) strcat(p, "s");
        else strcpy(p, "ies");
    else if (*p == 'z')
        if (isvowel(*q)) strcat(p, "zes");
        else strcat(p, "es");
    else strcat(p, "s");
}
```

Two of the tests depend upon whether the second from last letter is a vowel, so we define a little boolean function isvowel() for testing that condition:

```c
bool isvowel(char c)
{ return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u');
}
The test driver repeatedly reads a word, prints it, pluralizes it, and prints it again. The loop terminates when the user enters a single blank for a word:

```c
bool pluralize(char*);
int main()
{
char word[80];
for (;;)
{
cin.getline(word, 80);
if (*word == ' ') break;
cout << "\tThe singular is [" << word << "]..\n";
pluralize(word);
cout << "\tThe plural is [" << word << "]..\n";
}
}
```

8.5 We assume that names have no more than 25 characters and that there will be no more than 25 names. We’ll read all the input in at once and store it all in a single buffer. Since each name will be terminated with a NUL character, the buffer needs to be large enough to hold 25*(20 + 1) + 1 characters (25 21-character strings plus one last NUL character). The program is modularized into five function calls. The call input(buffer) reads everything into the buffer. The call tokenize(name, numNames, buffer) “tokenizes” the buffer, storing pointers to its names in the name array and returning the number of names in numNames. The call print(name, numNames) prints all the names that are stored in buffer. The call sort(name, numNames) does an indirect sort on the names stored in buffer by rearranging the pointers stored in the name array.

```c
#include <cstring>
#include <iostream>
using namespace std;
const int NAME_LENGTH = 20;
const int MAX_NUM_NAMES = 25;
```
const int BUFFER_LENGTH = MAX_NUM_NAMES*(NAME_LENGTH + 1);
void input(char* buffer);
void tokenize(char** name, int& numNames, char* buffer);
void print(char** name, int numNames);
void sort(char** name, int numNames);
int main()
{
    char* name[MAX_NUM_NAMES];
    char buffer[BUFFER_LENGTH+1];
    int numNames;
    input(buffer);
    tokenize(name, numNames, buffer);
    print(name, numNames);
    sort(name, numNames);
    print(name, numNames);
}

The entire input is done by the single call cin.getline(buffer, BUFFER_LENGTH, '$').
This reads characters until the "$" character is read, storing all the characters in buffer.

void input(char* buffer)
{
    // reads up to 25 strings into buffer:
    cout << "Enter up to " << MAX_NUM_NAMES << " names, one per"
    " line. Terminate with \"$\".\nNames are limited to "
    "NAME_LENGTH << " characters.\n"
    cin.getline(buffer, BUFFER_LENGTH, '$');
}

The tokenize() function uses the strtok() function to scan through the buffer, "tokenizing" each substring that ends with the newline character 'n' and storing its address in the name array. The for loop continues until p points to the sentinel '$'. Notice that the function's name parameter is declared as a char** because it is an array of pointers to chars. Also note that the counter n is declared as an int& (passed by reference) so that its new value is returned to main().

void tokenize(char** name, int& n, char* buffer)
{
    // copies address of each string in buffer into name array:
    char* p = strtok(buffer, "\n");        // p points to each token
    for (n = 0; p && *p != '$'; n++)
    { name[n] = p;
        p = strtok(NULL, "\n");
    }
}

The print() and sort() functions are similar to those seen before, except that both operate here indirectly. Both functions operate on the name array.

void print(char** name, int n)
{
    // prints the n names stored in buffer:
    cout << "The names are:\n";
    for (int i = 0; i < n; i++)
    { cout << \t<< i+1 << ". " << name[i] << endl;
    }
}

void sort(char** name, int n)
{
    // sorts the n names stored in buffer:
    char* temp;
    for (int i = 1; i < n; i++)                  // Bubble Sort
    for (int j = 0; j < n-i; j++)
        if (strcmp(name[j], name[j+1]) > 0)
        { temp = name[j];
        name[j] = temp;
        temp = name[j+1];
        name[j+1] = temp;
        }
    }
}
On this sample run the user entered 7 names and then the sentinel “$”. The names were then printed, sorted, and printed again.

8.6 The function first locates the end of the C-string. Then it swaps the first character with the last character, the second character with the second from last character, etc.:

```c
void reverse(char* s)
{
    char* end, temp;
    for (end = s; *end; end++)
        ; // find end of s
    while (s < end - 1)
    {
        temp = *--end;
        *end = *s;
        *s++ = temp;
    }
}
```

The test driver uses the `getline()` function to read the C-string. Then it prints it, reverses it, and prints it again:

```c
void reverse(char*);
int main()
{
    char string[80];
    cin.getline(string, 80);
    cout << "The string is [" << string << "]\n";
    reverse(string);
    cout << "The string is [" << string << "]\n";
}```
8.7

```c
int main()
{
    char word[80];
    while (cin >> word)
        if (*word) cout << "\t" << word << "\n";
}
```

8.8

```c
char* Strchr(const char* s, int c)
{
    for (const char* p=s; p && *p; p++)
        if (*p==c) return (char*)p;
    return 0;
}
```

8.9

```c
int numchr(const char* s, int c)
{
    int n=0;
    for (const char* p=s; p && *p; p++)
        if (*p==c) ++n;
    return n;
}
```

8.10

```c
char* Strchr(const char* s, int c)
{
    const char* pp=0;
    for (const char* p=s; p && *p; p++)
        if (*p==c) pp = p;
    return (char*)pp;
}
```

8.11

```c
char* Strstr(const char* s1, const char* s2)
{
    if (*s2==0) return (char*)s1;  // s2 is the empty string
    for ( ; *s1; s1++)
        if (*s1=='*s2)
            for (const char* p1=s1, * p2=s2; *p1==*p2; p1++, p2++)
                if (*(p2+1)==0) return (char*)s1;
    return 0;
}
```

8.12

```c
char* Strncpy(char* s1, const char* s2, size_t n)
{
    char* p=s1;
    for ( ; n>0 && *s2; n--)
        *p++ = *s2++;
    for ( ; n>0; n--)
        *p++ = 0;
    return s1;
}
```