```
runner% cat -n string_test.text
     1  runner% cat string_test.c
     2  #include <stdio.h>
     3  #include <string.h>
     4  #include <stdlib.h>
     5  void print_string1(char []);
     6  void print_string2(char *);
     7  int main()
     8  {
     9     /* three equivalent ways
              to declare the string "Neal" */
    10     char s[5];
    11     char t[] = "Neal";
    12     char u[] = {'N', 'e', 'a', 'l', '\0'};
    13     /* here is the address of a string: */
    14     char *p; char *q; char *r;
    15
    16     s[0] = 'N'; s[1] = 'e'; s[2] = 'a';
             s[3] = 'l'; s[4] = '\0';
    17     printf("Three strings:%s,%s,%s\n", s, t, u);
    18
    19     p = s; q = &s[0];
    20     printf("Using pointer to char:%s%s\n", p, q);
    21     /* passing as a parameter, all print: "Neal" */
    22     printf("Six Neals:");
    23     print_string1(s);
    24     print_string1(p);
    25     print_string1(q);
    26     print_string2(s);
    27     print_string2(p);
    28     print_string2(q);
    29     printf("\n");
    30     /* printing characters */
    31     printf("Four characters (1):%c%c%c%c\n",
    32        s[0], s[1], s[2], s[3]);
    33     printf("Four characters (2):%c%c%c%c\n",
    34        p[0], p[1], p[2], p[3]);
    35     printf("Four characters (3):%c%c%c%c\n",
    36        *s, *(s+1), *(s+2), *(s+3));
    37     printf("Four characters (4):%c%c%c%c\n",
    38        *p, *(p+1), *(p+2), *(p+3));
    39     printf("Four characters (5):");
    40     while (*p)  /* same as  while (*p != '\0')  */
    41        printf("%c", *p++);  /* same as *(p++)  */
    42     printf("\n");
    43     r = (char *) malloc(strlen(s) + 1);
    44     strcpy(r, s);
    45     printf("New string:%s\n", r);
    46     return 0;
    47  }
    48  void print_string1(char a[])
    49  {
    50     printf("%s", a);
    51  }
    52  void print_string2(char *a)
    53  {
    54     printf("%s", a);
    55  }
    56  runner% cc -o string_test string_test.c
    57  runner% string_test
    58  Three strings:Neal,Neal,Neal
    59  Using pointer to char:NealNeal
    60  Six Neals:NealNealNealNealNealNeal
    61  Four characters (1):Neal
    62  Four characters (2):Neal
    63  Four characters (3):Neal
    64  Four characters (4):Neal
    65  Four characters (5):Neal
    66  New string:Neal
runner%
```

Notes and Comments:

Lines 3-4: We need **<string.h>** for **strlen** and **strcpy**, and **<stdlib.h>** for **malloc**.

Lines 5-6: These two function prototypes are essentially identical, since as parameters, **char []** and **char *** are the same. In a prototype, we don't need the name of the parameter, though we could have written **char a[]** and **char *a**.

Line 10: This lays out a **char** array of size 5, with uninitialized (garbage values) stored in it.

Lines 11-12: If we initialize a string, either with **"???"** or with **{'?', '?', '?', '\0'};** then C will decide how long to make the array. Notice that in the first way, C puts in the **'\0'** char, while in the second way you have to do it yourself.

Line 14: The variables **p**, **q**, and **r** are declared to be of type "pointer to **char**", or "address of **char**". This is very similar to the type of variables **s**, **t**, and **u** , except that these latter are *constant* pointers to **char**. Thus **p = s** is legal, since **p** is not a constant, but **s = p** is illegal, since **s** cannot be changed (cannot be on the left side of an assignment statement).

Line 16: This is another (hard) way to initialize a character string. Note that it also is a true string, since I put the `'\0'` on at the end.

Line 17: Here I show that all three strings print out fine with a `%s` format. `%s` expects to see a variable of type `char *` (pointer to `char`) later in the `printf` statement.

Line 19: `p = s;` just puts the address of `s` into `p`. Now `p` will behave in many respects like `s`, except that I can change `p` again if I like. The other `q = &s[0];` is much trickier, but does the same thing. `s[0]` is the first element of the array `s` (the first character), and `&s[0]` is the address of that first char, so that `&s[0]` is just a fancy way to write `s`, a pointer to the start of the character string.

*Line 20:*Here I'm showing that `%s` works fine with `p` and `q`, when they've been initialized correctly, as well as `s`, `t`, and `u`.

Lines 22-29: These call the two functions in various ways to print the character strings. As you see, any combination of `char []` or `char *`, passed to either of the two functions, works fine.

Lines 30-42: These show 5 *equivalent* ways to print the four characters in the various strings. Each is printed with a `%c` format, so the variable is supposed to be just of type char.

Lines 31-32: This just prints the four array elements in a straightforward way.

Lines 33-34: This shows that even for something like `p` that was declared of type `char *`, the `[]` subscript notation still works (of course assuming that `p` has been initialized to the address of an actual character string).

Lines 35-38: This shows the "pointer arithmetic". Given an address like `s` or `p`, we can write `s+1` or `p+1` for the next item pointed to. (`s` or `p` is the address of the zeroth item in the array, while `s+1` or `p+1` points to the first item, and `s+2` or `p+2` is the address of the second item, and so forth. Given the address of something, in C, the `*` operator fetches what is at that address (we say "dereference"). Thus `s[0]` is the same as `*s`, and `s[1]` is the same as `*(s+1)`, `s[2]` is the same as `*(s+2)`, and so forth. Also the same is true for `p`. In fact, C just translates any expression like `p[2]` into the equivalent form `*(p+2)`. Notice that `*p+2` which is the same as `(*p)+2` is something completely different. This last will add `2` to the *value* of `*p`.

Lines 39-42: This is similar, except that we are actually incrementing the value stored in `p`. Here we must have a variable like `p` that can be changed, rather than `s`. When I first wrote this segment, the while loop was `while(p != 0)` instead of `while(*p != 0)`, which is correct. (The incorrect version produced a segmentation error.) The correct version lets `p` be incremented until it points to the null character `'\0'` at the end of the character array. We can also write `while (*p)`. The incorrect version starts with an non-zero address stored in `p` and just increments it indefinitely, so of course it will never be zero. Notice that this little segment destroys the value of `p`, since when it is done, `p` points to a null character, and the string is no longer accessible through `p`. If instead we wrote a separate function, with `p` passed by value as a parameter, then this would work fine.

Lines 43-45: This is the most sophisticated code here. The declaration `char *r;` creates a location `r` that is ready to hold the address of a char (of the starting address of a string). Initially, `r` will have useless garbage stored in it. The function `malloc` allocates storage at run time and returns the address of this storage. `malloc(strlen(s) + 1);` will allocate room for 5 characters in this case, enough for the non-null characters in `s`, and one more for the null. The address of this storage is stored in the variable `r`. Then, the `strcpy` function copies the characters in `s` into the new storage in `r`. (The `'\0'` at the end is copied also.) Finally, `r` can be printed like any other string. This method of using `malloc` is the most common and flexible way to work with strings in C.