Consider the following complete program:

```
1  #include <iostream>
   using namespace std;
3
   char* add_size(const char* s) {
5      if (s==0) return 0;
       int n = 0;
7      while (s[n] != 0)
           ++n;
9      char* pc = new char[n+2];
       for (int i = 0; s[i]; ++i)
11         pc[i] = s[i];
       pc[n] = 0;
13     delete[] s;
       return pc;
15 }

17 int main() {
       char ch;
19     char* my_array = new char[1];
       *my_array = 0;
21     while (cin>>ch && ch!='!') {
           my_array = add_size(my_array);
23         int i = 0;
           for (; my_array[i]; ++i);
25         my_array[i] = ch;
           my_array[i+1] = 0;
27     }

29     cout << my_array << "\n";
       delete[] my_array;
31 }
```

1. What is the purpose of the `const` on line 4?

   ANS: The function `add_size` takes a parameter, `const char* s`, which is a pointer. As such, it specifies the memory address of a different memory location and the type of thing it is pointing to (the number of bytes required for one of these) as a value, which it names `s`, a pointer to a constant character. It is not a constant pointer, so you could change what it's pointing to. Examples with comments:

   ```
   *s = 'a'; // wrong:   s points to a const char
   s++;    // right:  the pointer itself is not constant
   ```

   The reason for making the argument c-string constant is because we just want to copy it and make it one character longer. We actually delete it after copying it into a longer vector.

   Note that here the value of the pointer is passed to the function (by value) and can then be derefenced–this is sometimes called, "C style pass-by-reference."

2. Why are `n+2` bytes allocated for `pc` on line 9?

   ANS: The key to understanding this is in the name of the function. Good function naming creates self-documenting code. The name, `add_size` suggests that we want to add to the size of the input string. So why not make it one larger with `n+1` bytes? The code fragment,

```
int n = 0;
while (s[n] != 0)
    ++n;
```

will loop until `n` is the number characters in the sequence of bytes (characters) whose starting address is `s` the `NULL` character, but the string also includes the `NULL` character, so it will be `n+1` characters long before `add_size()` and, to make it 1 longer, we make allocate memory on the free store for 1 more than the `n+1` we had before.

3. Why assign `pc[n]` to `0` on line 12?
   To see this, you really need to understand the flow of the program. Before entering the `while` loop in `main()`, `my_arry` is declared as a pointer to a `char` and allocates enough memory for one `char` (one byte), initializing the pointer to point at this byte. Then `my_array` is dereferenced and its value set to 0 (NULL.)
   Let's say the user enters `'dog'`, then in the `while` loop, `add_size` is called `my_array`, and since `my_array` contains only the `NULL` character, `add_size` just returns the `NULL` character. Then `i` is set to zero and the `for` loop doesn't execute (`my_array` is just the `NULL` character, so the condition for looping is `false`) and `ny_arra[0]` is set to `'d'` followed by `my_array[1]` containing the `NULL` character (yikes! Assigning a value to unallocated memory!). Now `cin` fetches `'o'` from the keyboard buffer and `add_size` is called on `my_array` again, this time local variable `n` gets to count the current size of the c-string (1) and and so the local pointer to `char, pc` is created and made to point to the first of enough memory to hold 3 characters. The `for` loop that follows copies characters from the memory at `my_array` until a `NULL` is encountered. Then the 2nd (`n+1th`) byte is set to `NULL`, the local memory is freed and the pointer `pc` is returned.

   So why allocate memory for three bytes when only two are used at this point? Because on line 26, we're going to move the `NULL` character to the third byte, and now we do that without writing to unallocated memory!

4. Describe, in detail, what the `while` loop of `main()` is doing.
   See #3. for much of this detail. The condition for looping is the `and` of two `bool`s: (1) did the user successfully enter a character? and (2) was that character not `'!'`? If both of those `bool`s are `true`, then `my_array` is passed to `add_size()`, which copies the c-string, adding 1 to the size and puts the `NUL` character in the `n+1` position (index `n`) and returns the copy back to `my_array`. Lines 23,24 then use a for loop to find the position of the `NUL` character and replace it with the character read into `ch`, moving the `NUL` character to the next position.

5. What is the output of the `main()` if the user enters "cattdog"?
   Well, nothing, since the character `'!'` is not encountered. If we entered `'catdog!'` then the program would just echo back our input and quit.