CS 7B - Fall 2017 - Pointer and Reference Quiz 01

Consider the following complete program:

```
1  #include<iostream>
   using namespace std;
3
   char* strcopy(const char* s) {
5      if (s==0) return 0;

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

11     char* pc = new char[n+1];

13     for (int i = 0; s[i]; ++i)
           pc[i] = s[i];
15     pc[n] = 0;

17     delete[] s;
       return pc;
19 }
   void* address(void* input) {
21     return &input;
   }
23 int main() {
       string s;
25     char* cstr;
       while (cin>>s && s!="quit") {
27         cstr = strcopy(&s[0]);
           cout << cstr << "\n";
29         cout << address(cstr) << endl;
           cout << address(&*(cstr+2));
31         delete[] cstr;
       }
33 }
```

1. Explain what the conditional on line 5 is doing.
   ANS: Check whether the string passes to `strcopy` is the empty string. If so, return the `NULL` pointer, 0.

2. What is the while loop on line 8 doing?
   ANS: It finds the length of the string `s` by looking at the `char` pointer of the n=1st element at comparing it with `NULL` pointer, incrementing n until the `NULL` pointer is found. When it's done n contains the length of the string `s`.

3. What happens on line 11?
   ANS: Enough memory is allocated to accomodate $n+1$ bytes of type `char` and the memory address of the first of these is stored in a variable named `pc`.

4. What's happening on line 13?
   ANS: Starting at `i=0`, while `s[i]` is not the `NULL` character, it's value is assigned to the newly created memory at `pc[i]`.

5. What is the purpose of line 17?
   ANS: Deallocate the memory allocated for `s`.

6. What is the purpose of line 18?
   ANS: Return the address of the memory copied from `s` to the calling function.

7. What is the `address()` function do?
   It takes a pointer variable (a pointer to any type) and converts it to a pointer to `void` (`void*`) named `input` and

returns the address of the thing pointed to by that pointer. This means the unusual behavior of dereferencing a character pointer is defeated and we get the address instead of the contents of that address.

But there is an error in `address()`: it returns the address of the local variable created rather than the address of pointer passed.

8. If the user types "dog" into the keyboard and presses "Enter" what will the output be? (Just describe it...)
ANS: `cin` will insert "dog" from the keyboard buffer into the `string` variable `s` and since this happens successfully, and the string is not the same as "quit", we enter the `while` loop which calls `strcpy` on `&s[0]` (which by the way is the same as calling `strcpy` on `s`). `strcpy` then copies the characters of `s` into `pc` and places a terminating NULL at the endn of `pc` and returns the pointer to `char pc` to the caller. Since we've entered "dog", this is printed to console, followed by the call to print the output of `address(cstr)`, ***which was intended to be*** the memory address of `cstr`. I then print the output of `address(cstr+2)`, but, strangely, that is the same as the address of `cstr`:

```
dog
dog
0x28fea0
0x28fea0
```

What's going on?! Read on... The reason the same address is printed twice is because `address()` is returning `&input` which is the address of the place where the local variable `input` is stored. If we change `address()` to return `input` instead of `&input`, we get output like this:

```
dog
dog
0x9a6c18
0x9a6c1a
```

...where the address of the third character in dog is clearly two bytes more than the address of first character.

To be sure, let's change the body of `address()` like so:

```
void* address(void* input) {
    cout << "\n input = " << input;
    cout << "\n&input = " << &input << endl;
    return input;
}
```

Then we can see what's happening more clearly: the variable `input` of type pointer to `void` holds the address of the parameter passed by value, which is the address of the characters in "dog", and itself has an address. Unsurprisingly, the compiler assigns the same memory address to input on both calls to `address()` in `main()`, but the second call passes the address of the third character instead of the first.
Here is output that results from this:

```
dog
dog

 input = 0x6b6c18
&input = 0x28fea0
0x6b6c18

 input = 0x6b6c1a
&input = 0x28fea0
0x6b6c1a
```

Finally, the memory for `cstr` is freed with `delete [] cstr`–though it will be deleted on exiting `main()` anyway.

Note that the reference and dereference operators `&*` on line 30 just cancel eachother out.