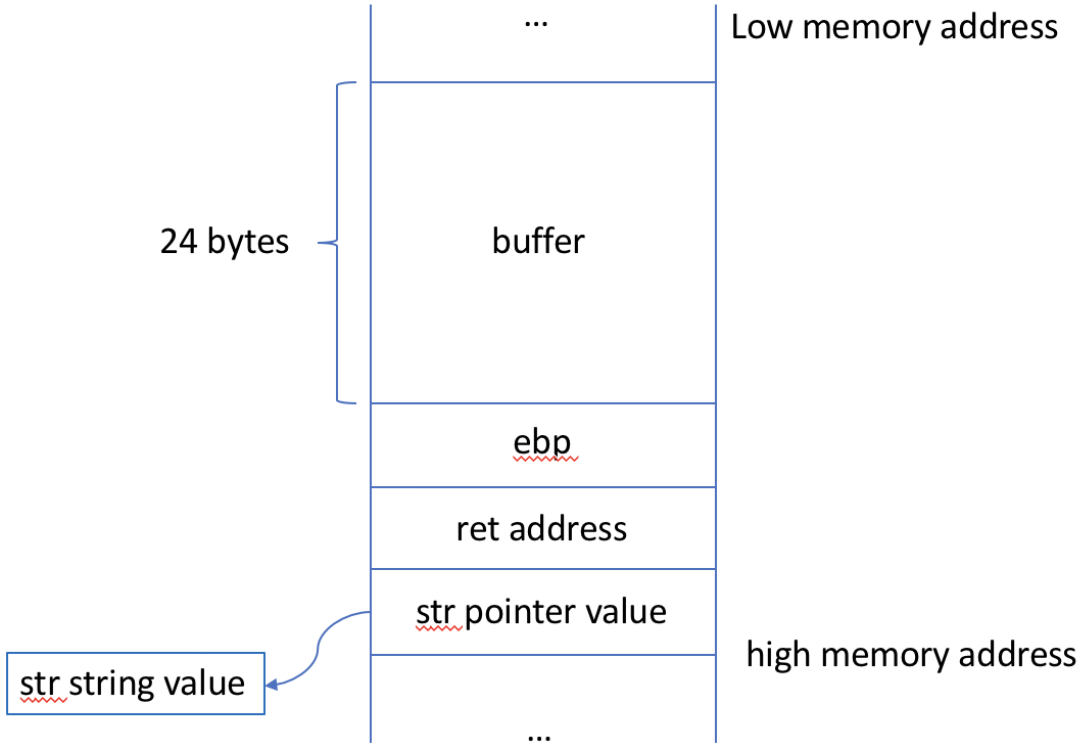
Buffer overflow
(optional assignment)

Question 1: Please draw the function stack frame for the following C function.

```
int bof (char *str) {
   char buffer[24];
   strcpy (buffer, str);
   return 1;
}
```

Answer:

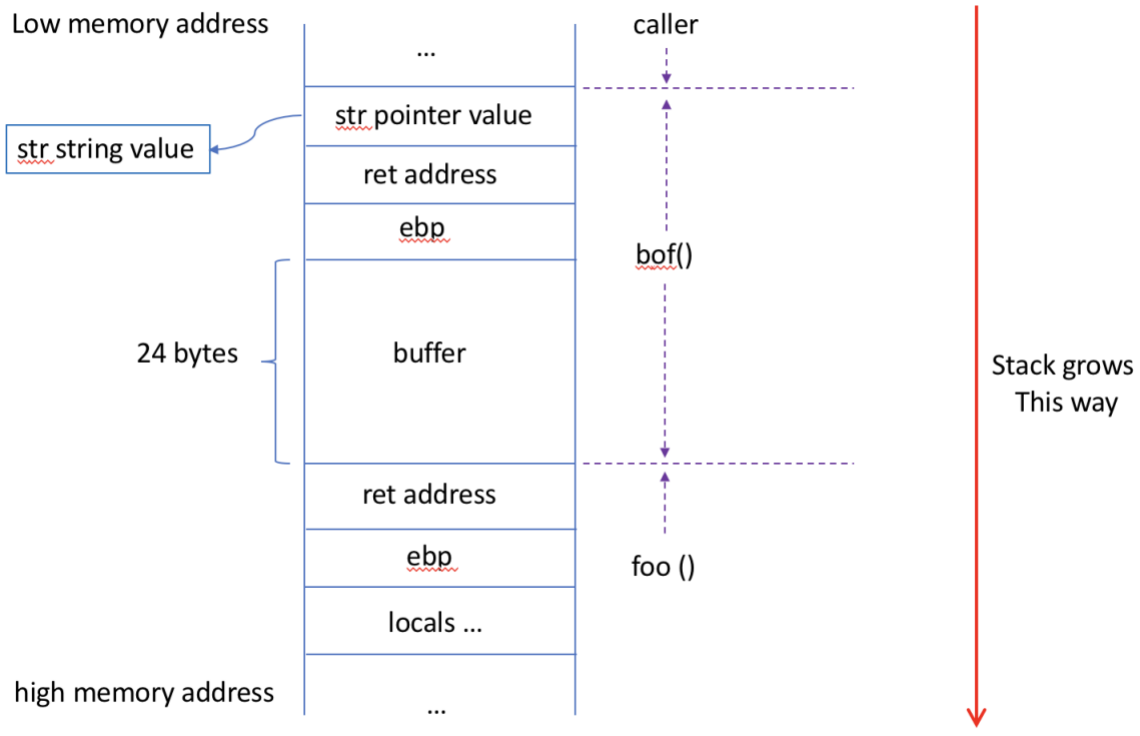


Question 2: A student proposes to change how the stack grows. Instead of growing from high address to low address, the student proposes to let the stack grow from low address to high address. This way, the buffer will be allocated above the return address, so overflowing the buffer will not be able to affect the return address. Please comment on this proposal.

Answer: This will still affect the return address. For example:

```
int bof (char *str) {
   char buffer[24];
   foo ();
   strcpy (buffer, str);
   return 1;
}
```

For the code segment above, overwriting the buffer in bof will affect the return address of the foo function. After the foo function is executed, the execution will still be jumped to the location set by the overflowing, attack-controlled data.



Question 3: Why does ASLR make buffer-overflow attack more difficult?

Answer: A buffer overflow is exploited by an attacker by trying to making the extra bytes spill over some other elements in a controlled way. ASLR makes that more difficult by "moving things around", the field that the attacker tries to overwrite will not necessarily be located at a predictable place beyond the overflowing buffer.

Question 4: The buffer overflow example was fixed as below. Is this safe?

```
int bof (char *str, int size) {
   char *buffer = (char *) malloc (size);
   strcpy (buffer, str);
   return 1;
}
```

Answer: No. The length of str is still possible to be larger than the value of the *size* parameter. To fix the buffer flow vulnerability, we need to use strlcpy: strlcpy (buffer, str, size);