

Lecture 06 – Format string vulnerabilities

Stephen Checkoway
University of Illinois at Chicago

Goal

- Take control of the program (as usual)
- How?
 - Write4 (write 4 bytes to an arbitrary location)
 - Inject shellcode (or other exploits) into the process

What should we overwrite?

- Saved instruction pointer (seip)
- Other pointers to code (we'll come back to this)

printf operation

- printf takes a format string and arguments
- printf copies the format string to its output, replacing conversion specifiers with values determined by the arguments
- Arguments are (normally) accessed one at a time in turn
- Internally, printf keeps a pointer to the next argument to be converted by a conversion specifier
- Example: `printf("value = %d %c", 42, 'm');`
prints: `value = 42 m`

Common conversion specifiers

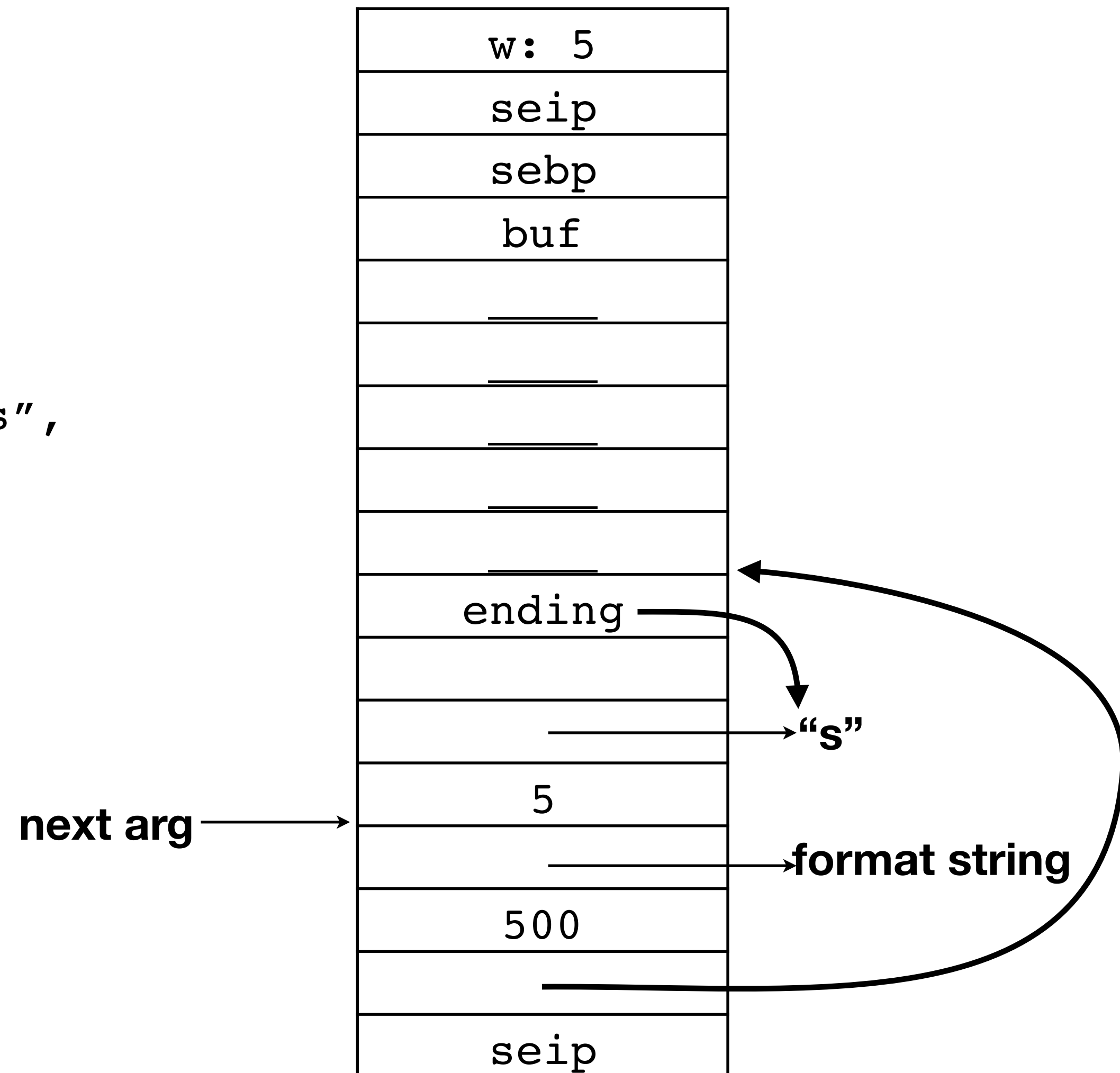
%c	Character	%s	String
%d, %i	Integer	%p	Pointer
%u	Unsigned integer	%%	Literal %
%x, %X	Hex	%n	Stores number of characters written
%e, %f,	Double		

printf family

- printf
- fprintf
- sprintf
- snprintf
- asprintf
- dprintf
- vprintf
- vfprintf
- vsprintf
- vsnprintf
- vasprintf
- vdprintf

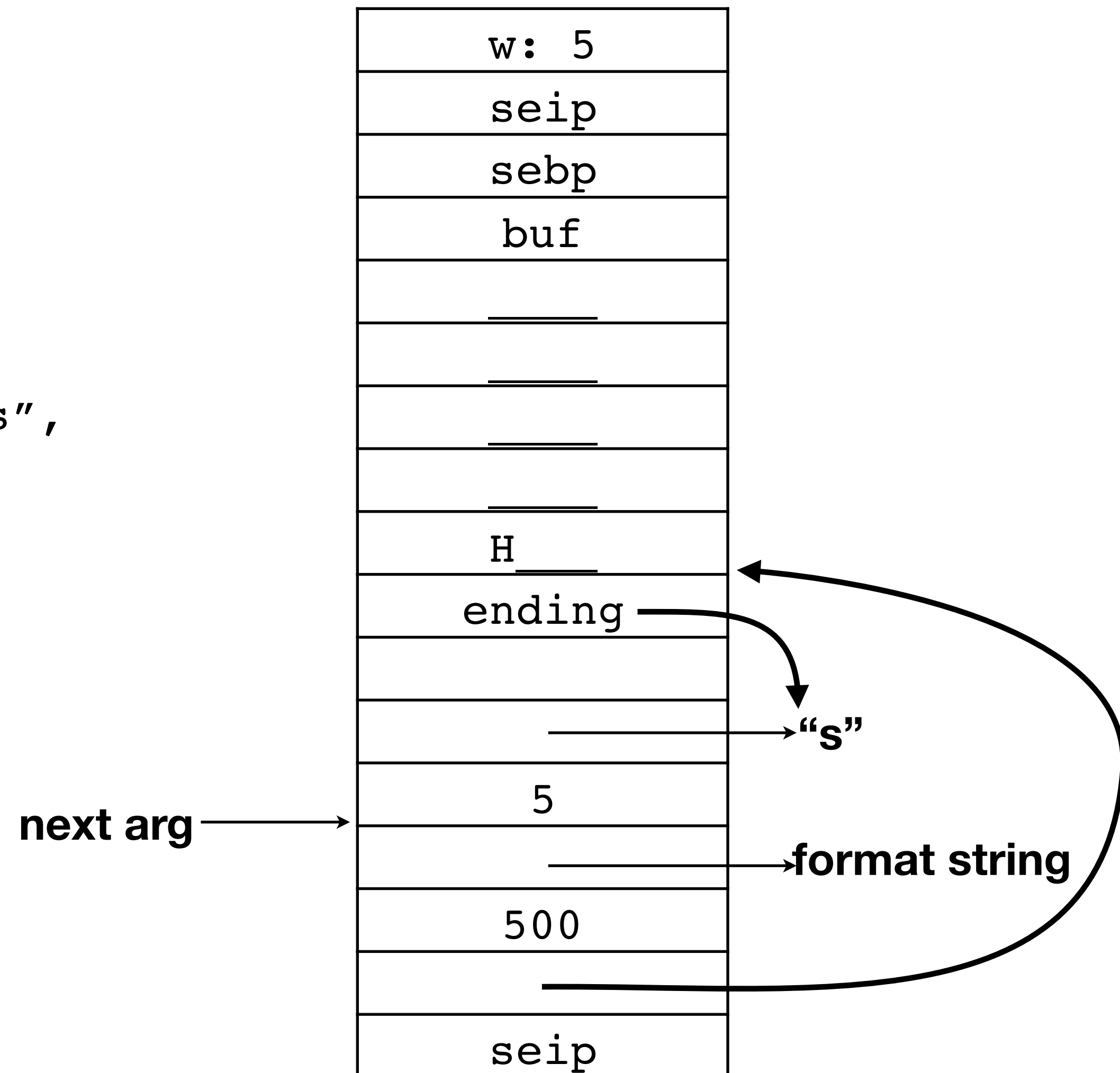
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



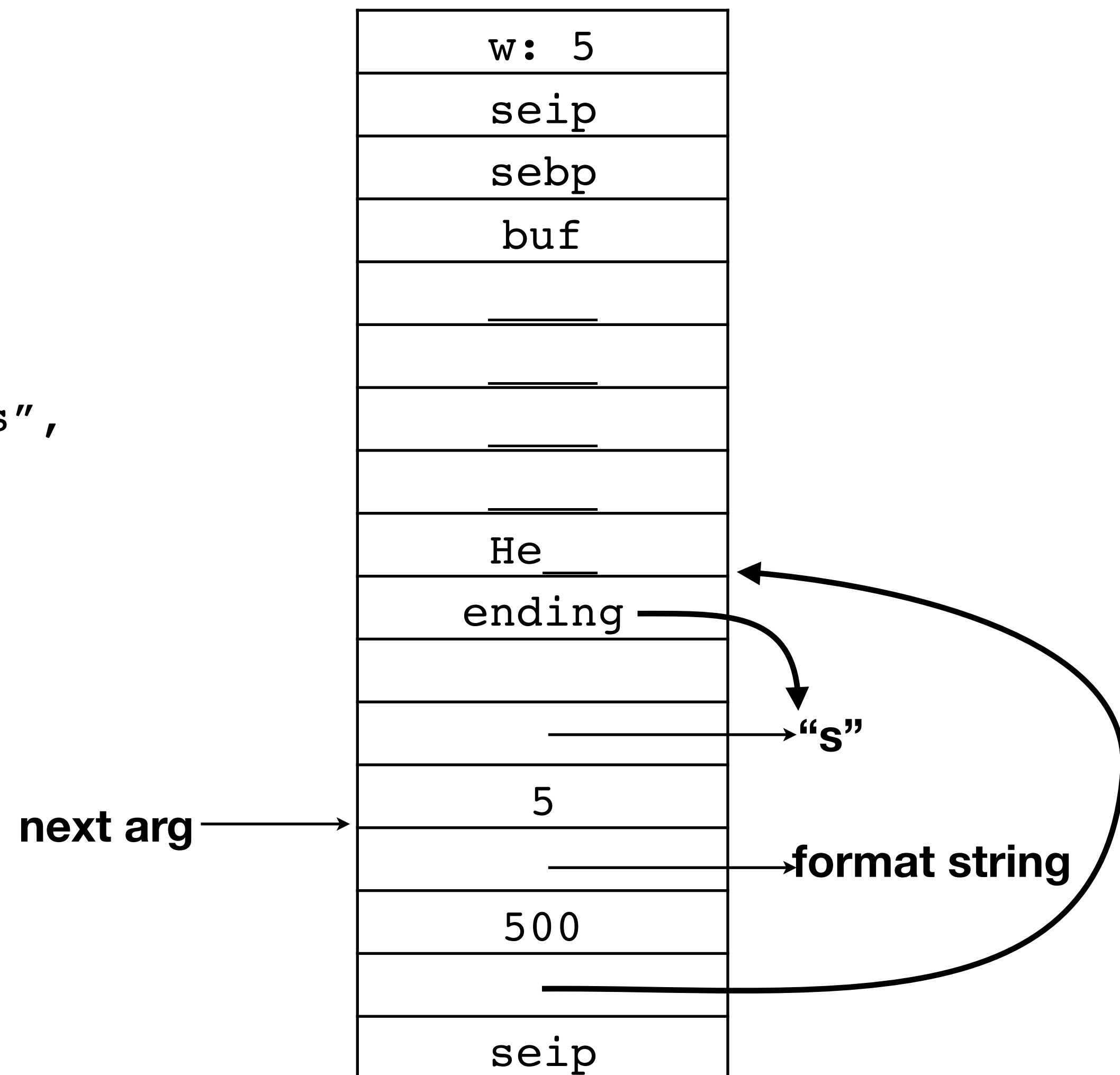
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



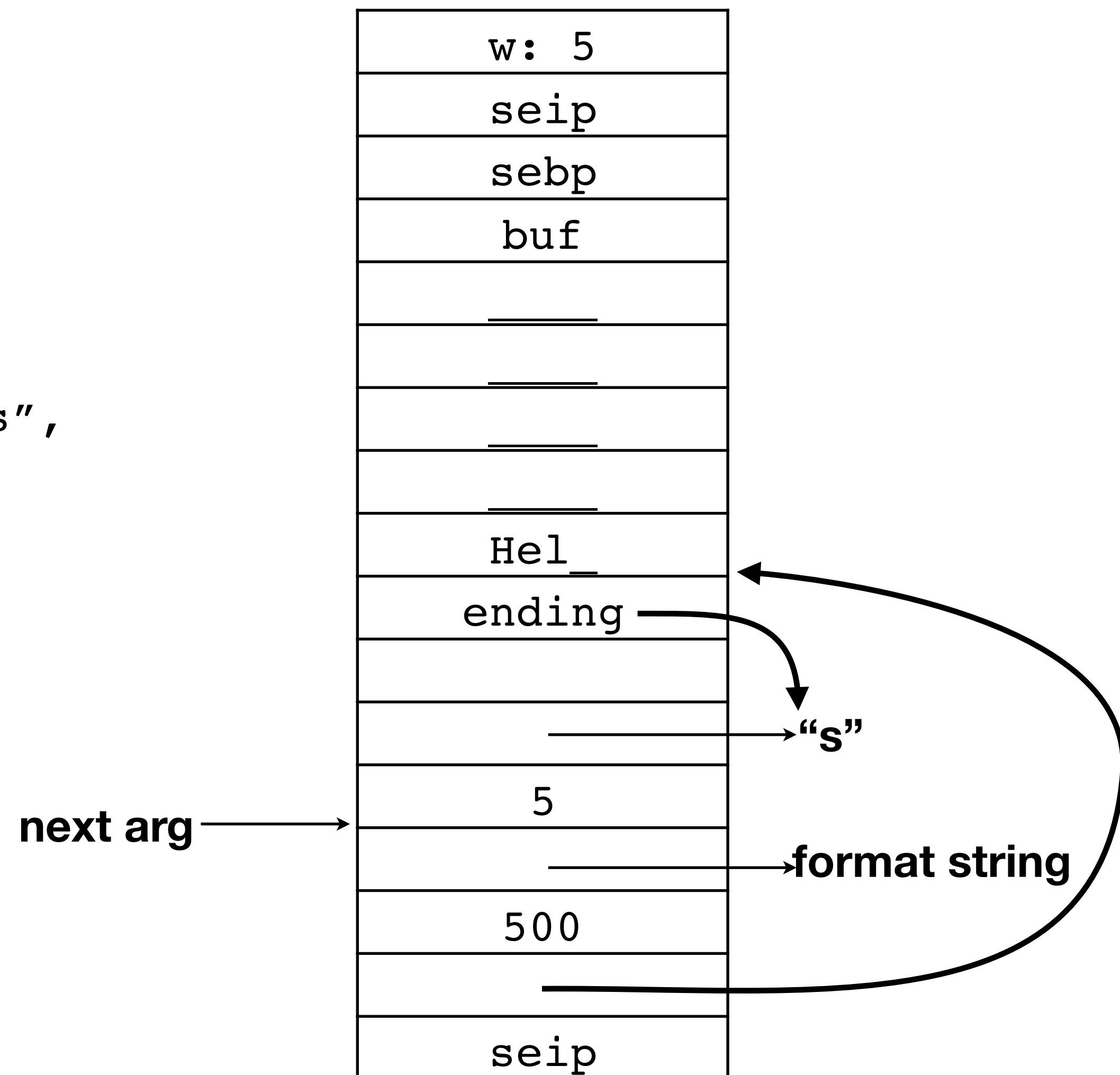
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



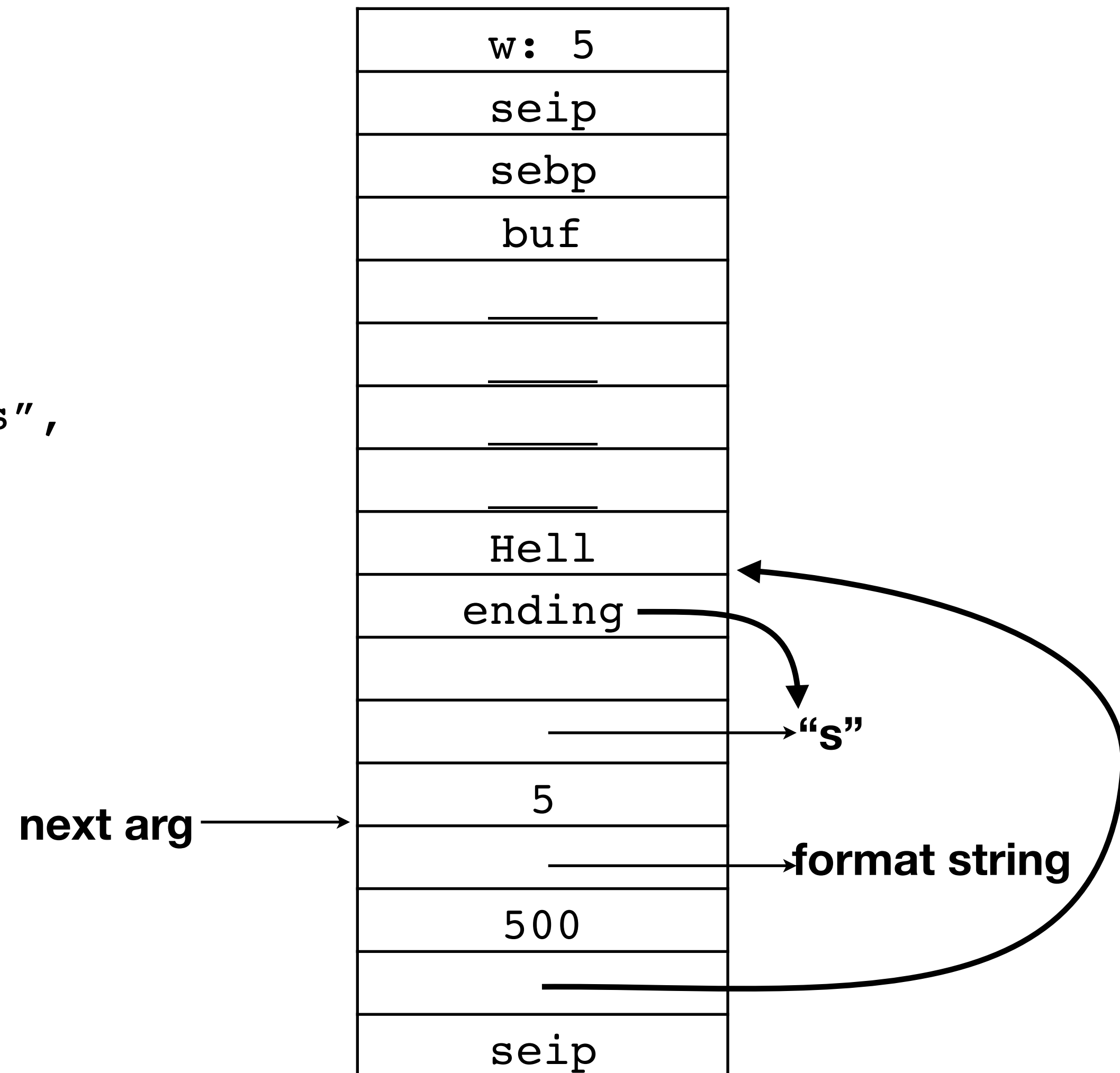
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



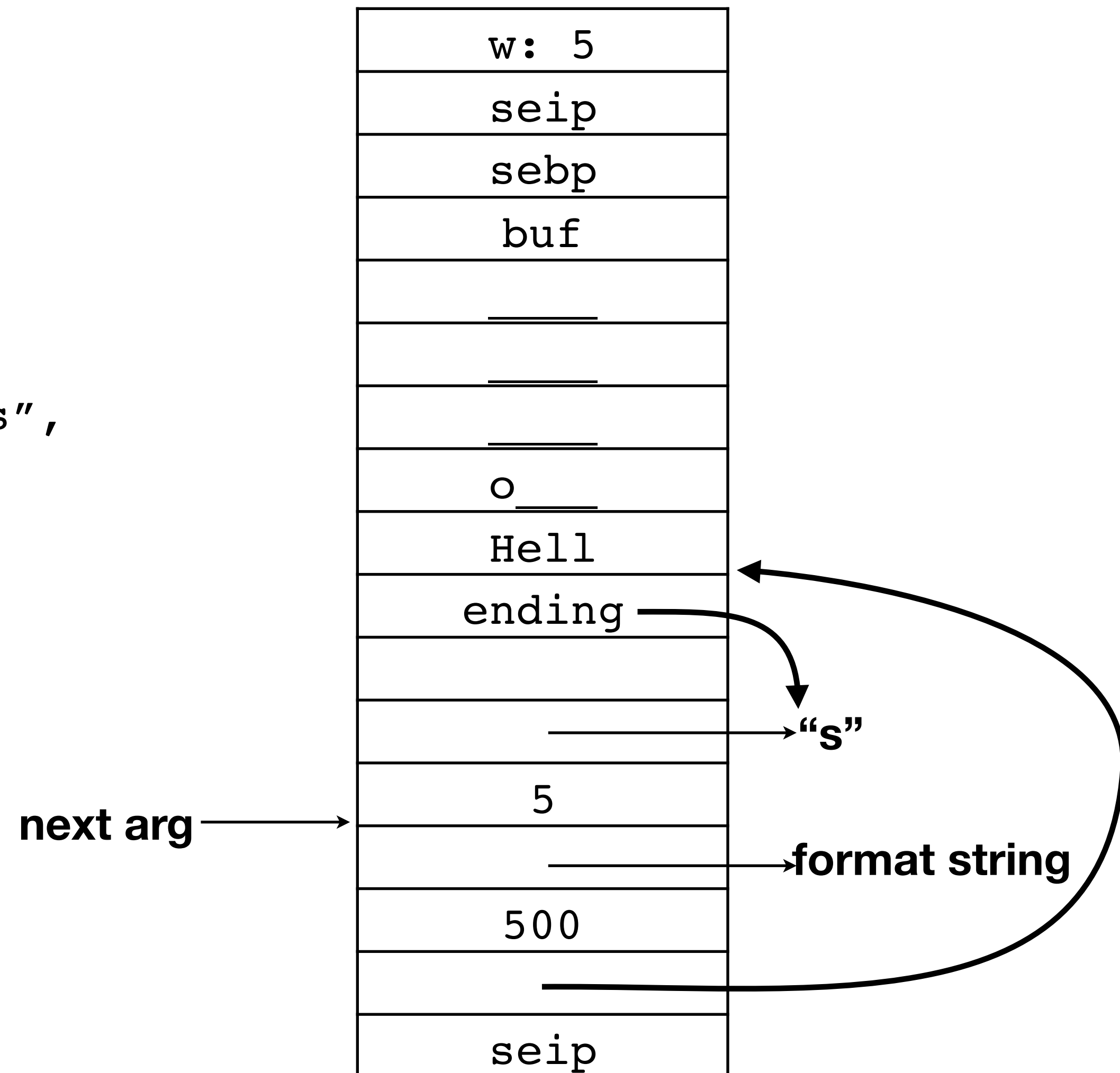
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



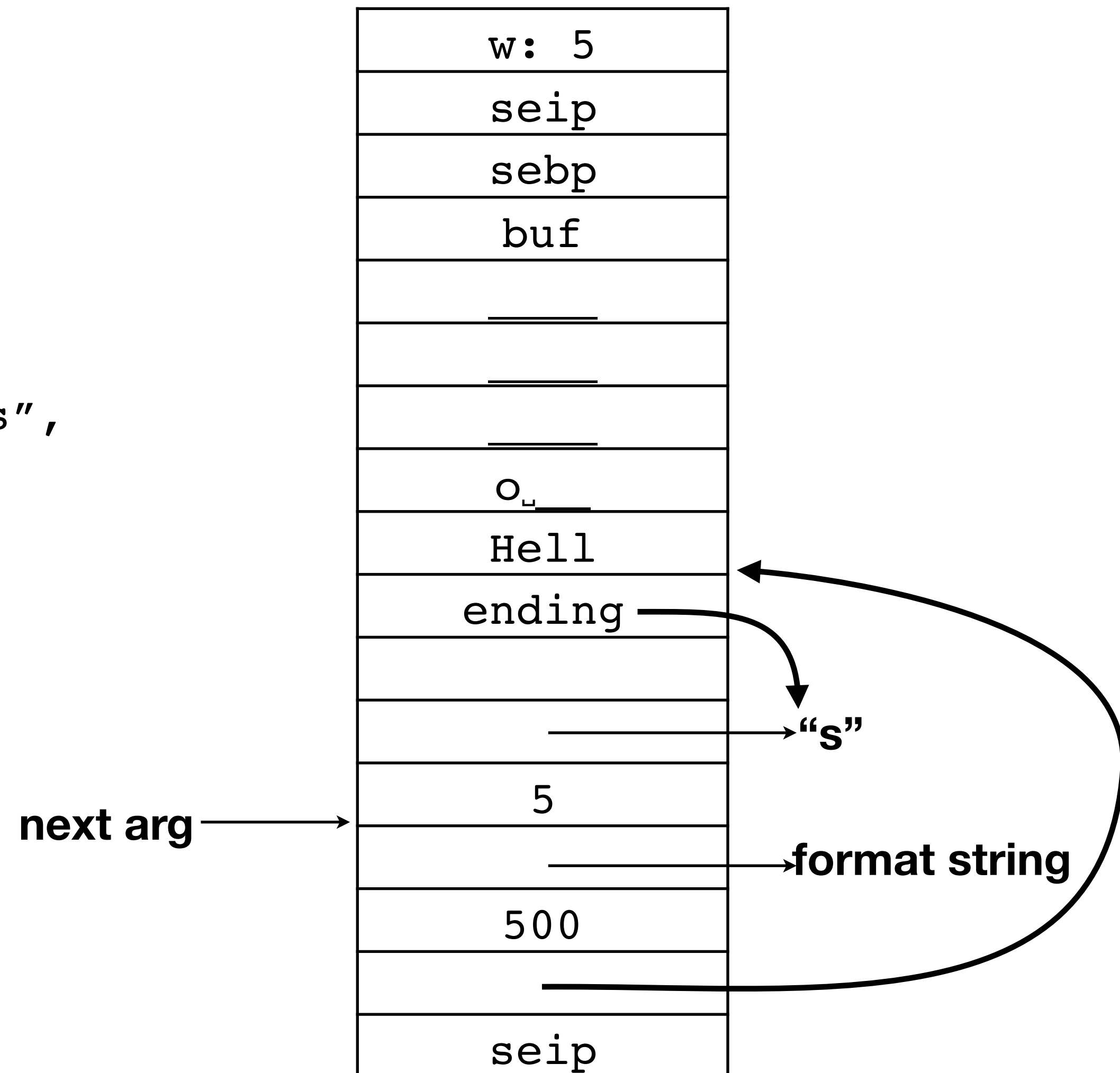
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



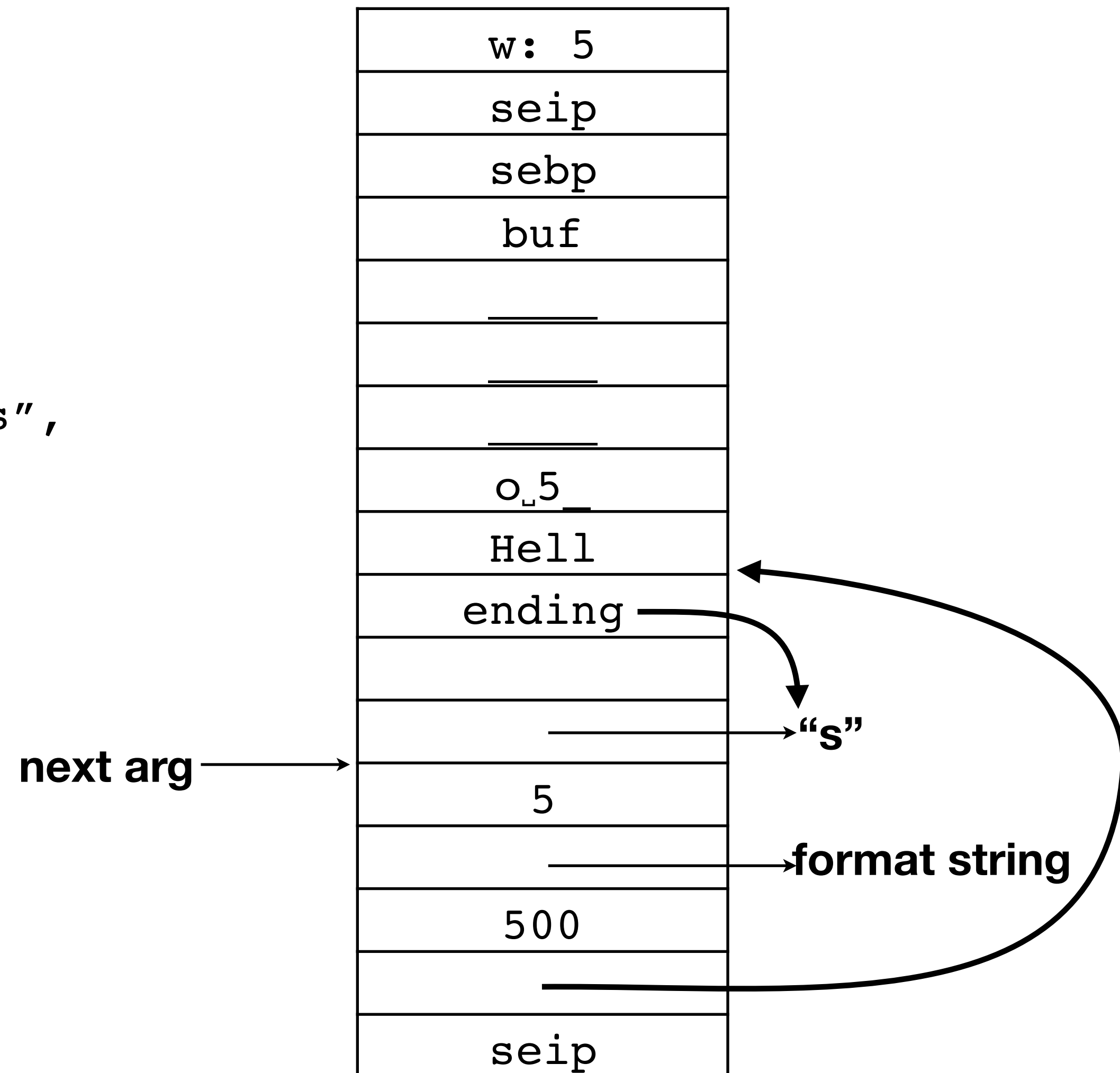
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



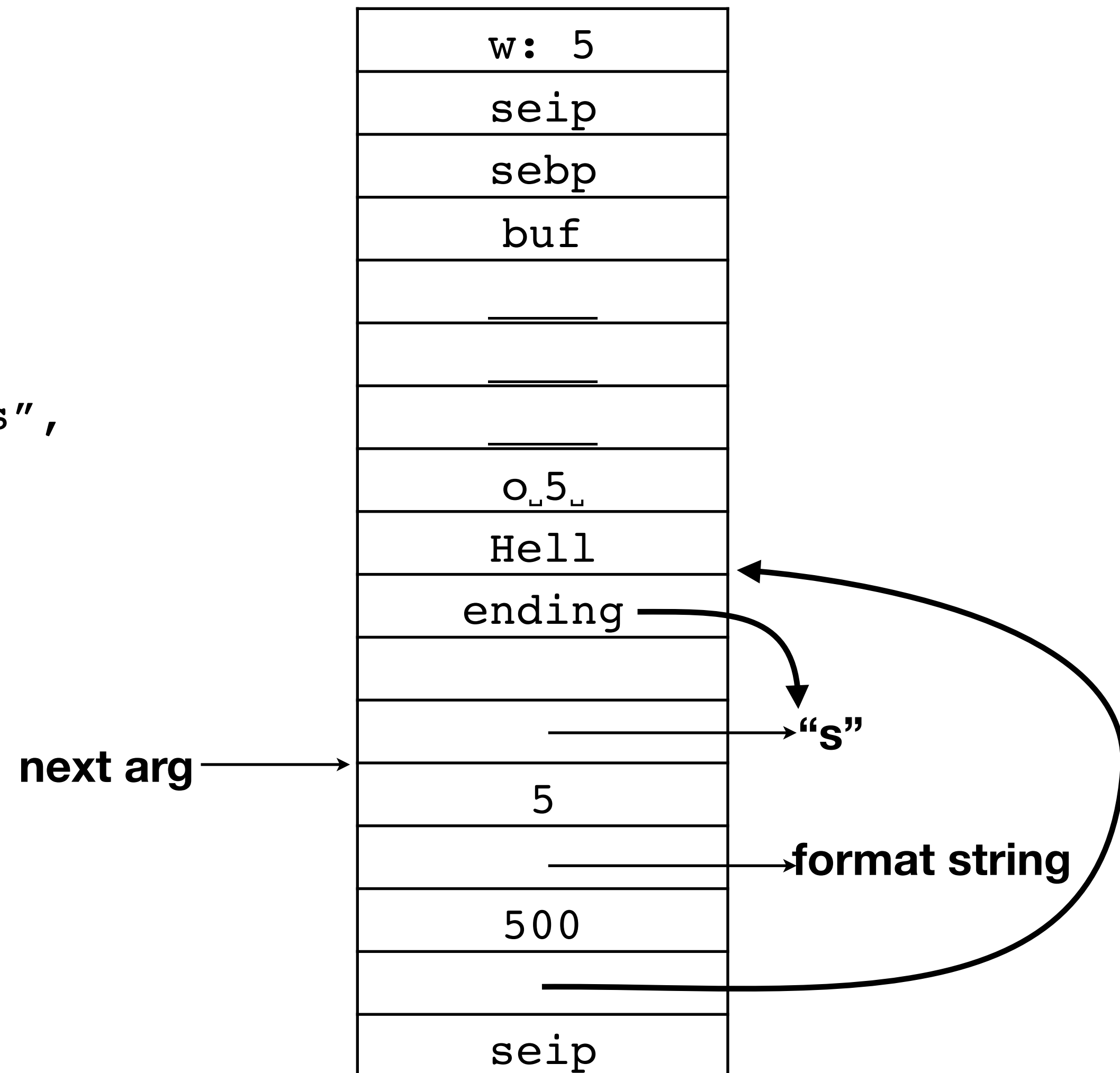
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



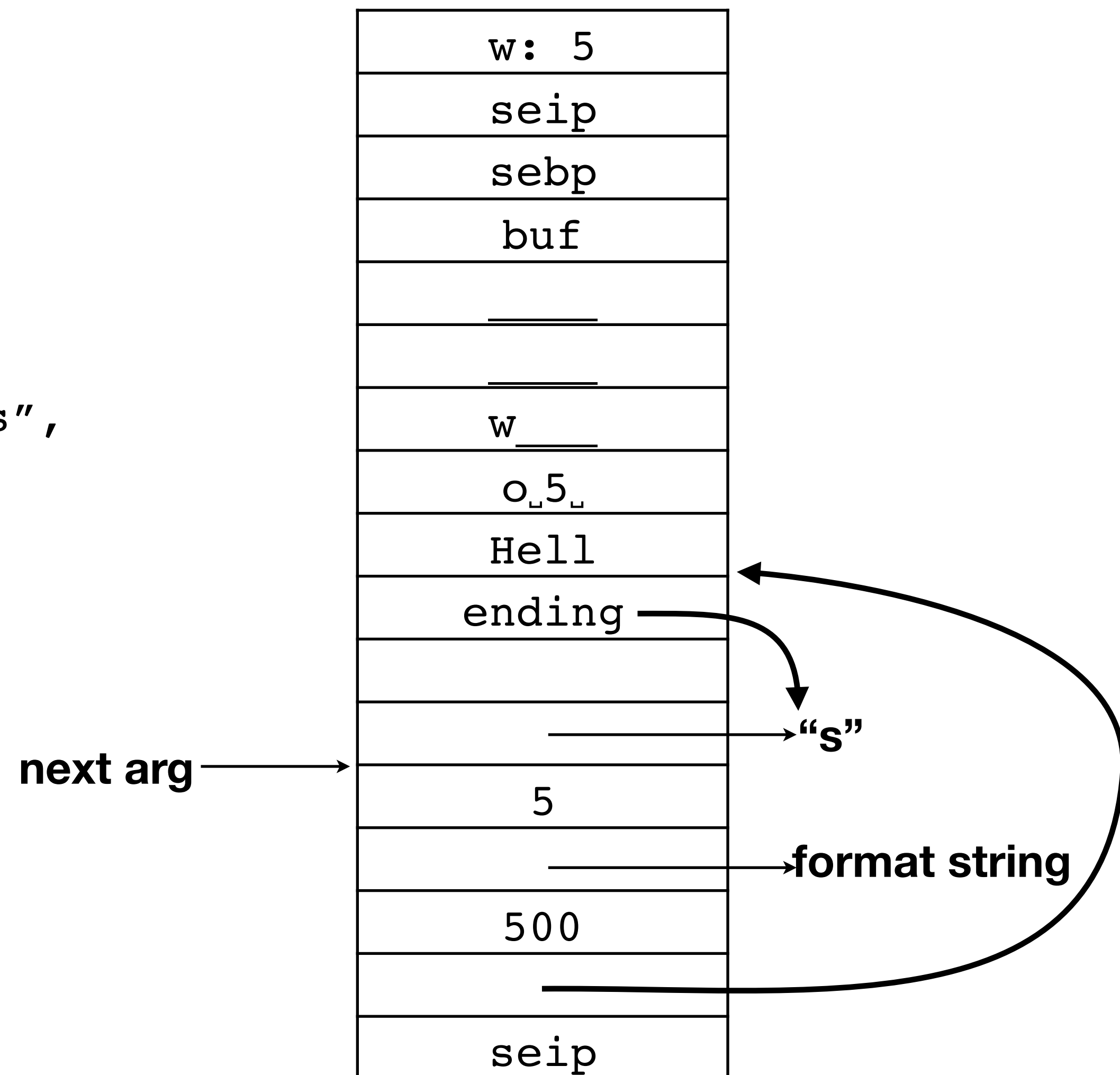
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



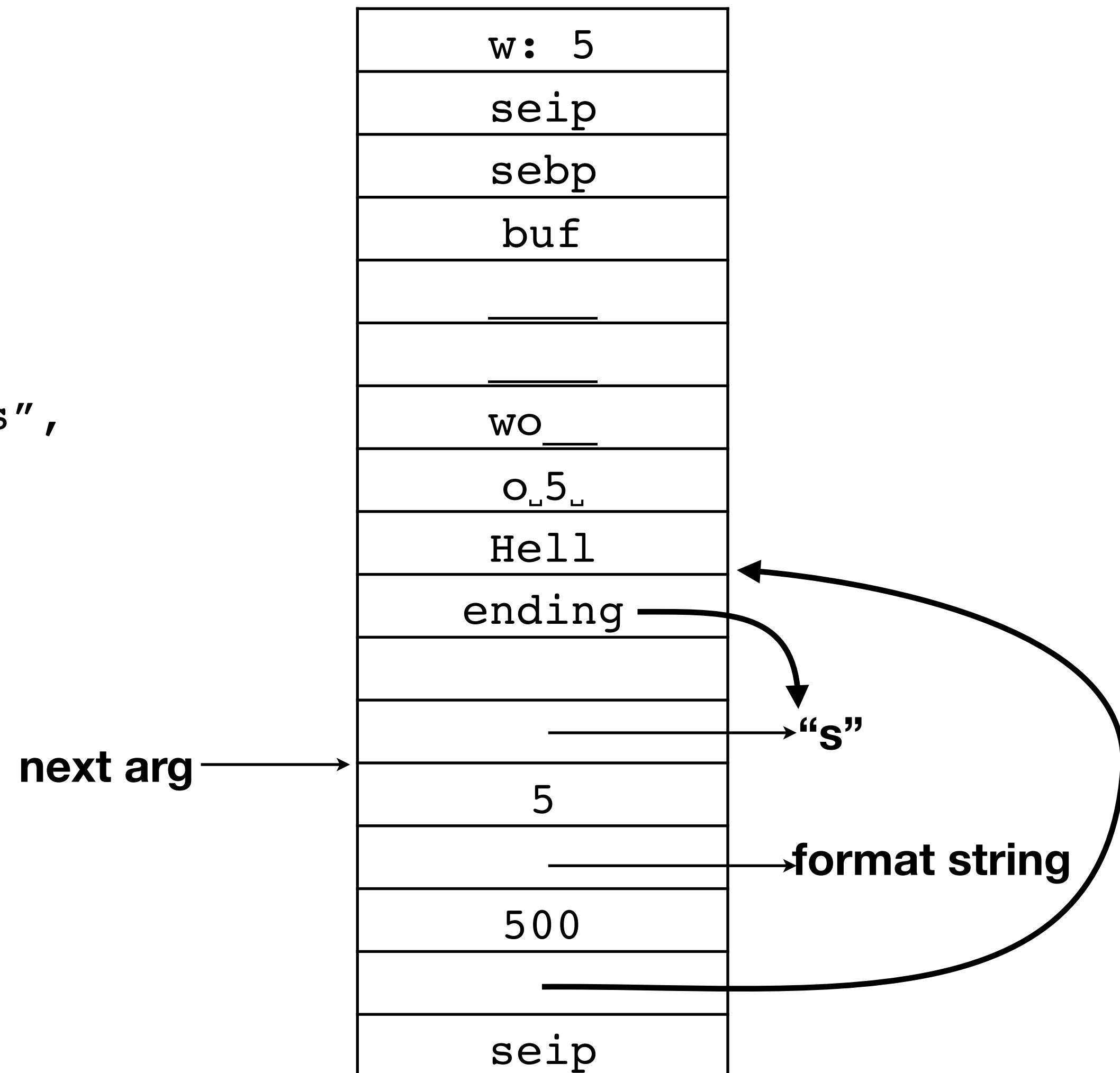
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



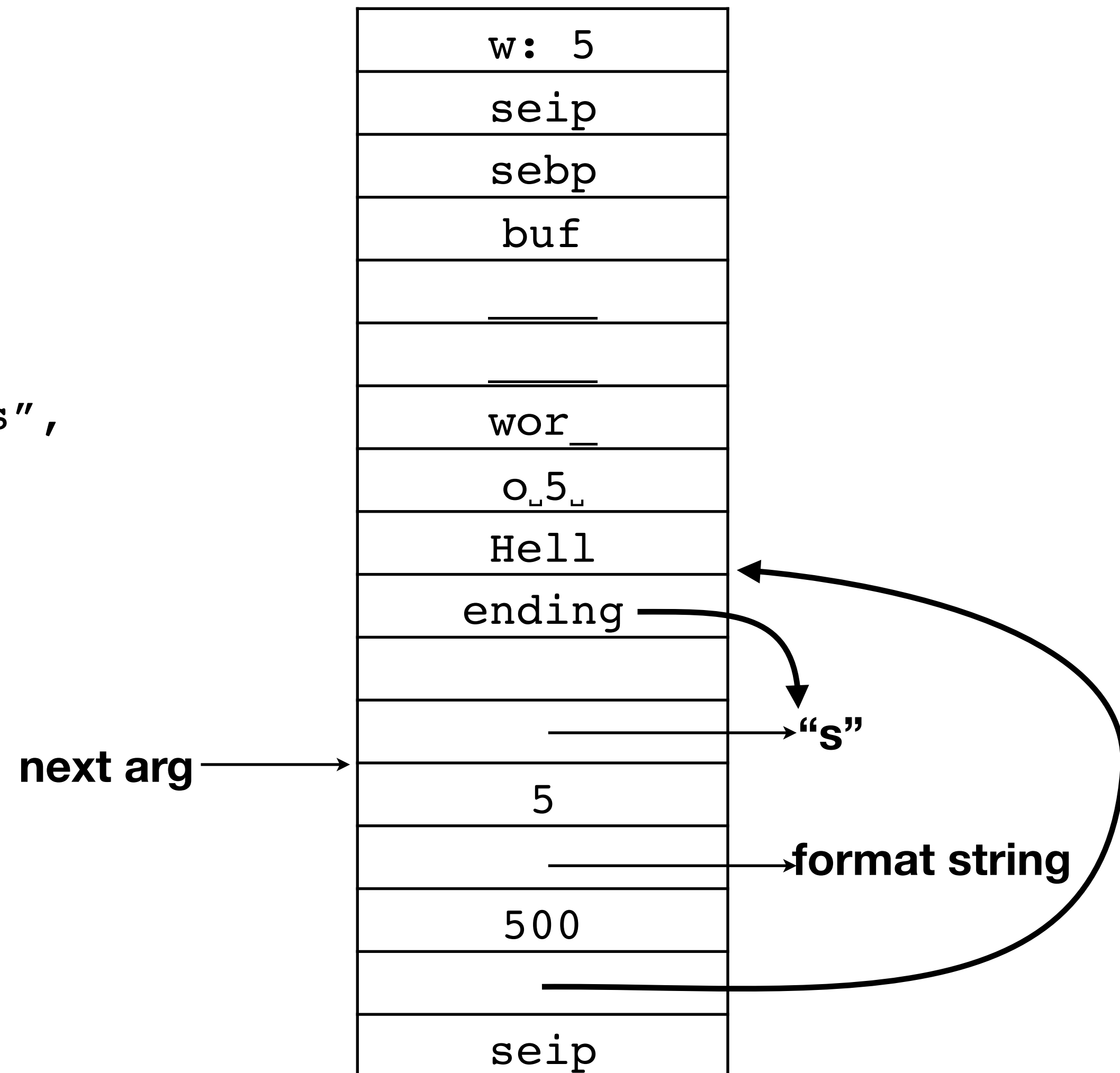
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



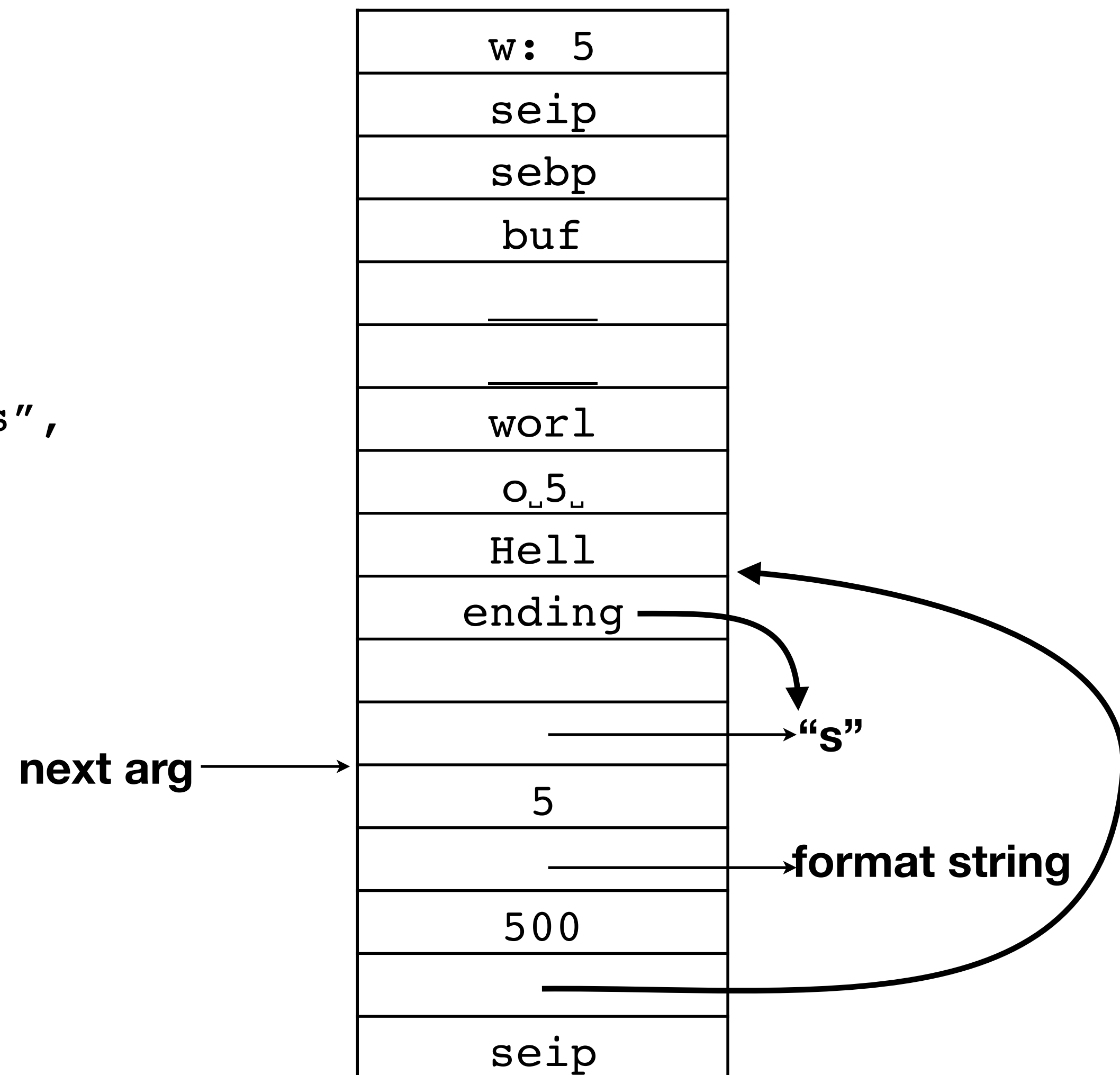
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



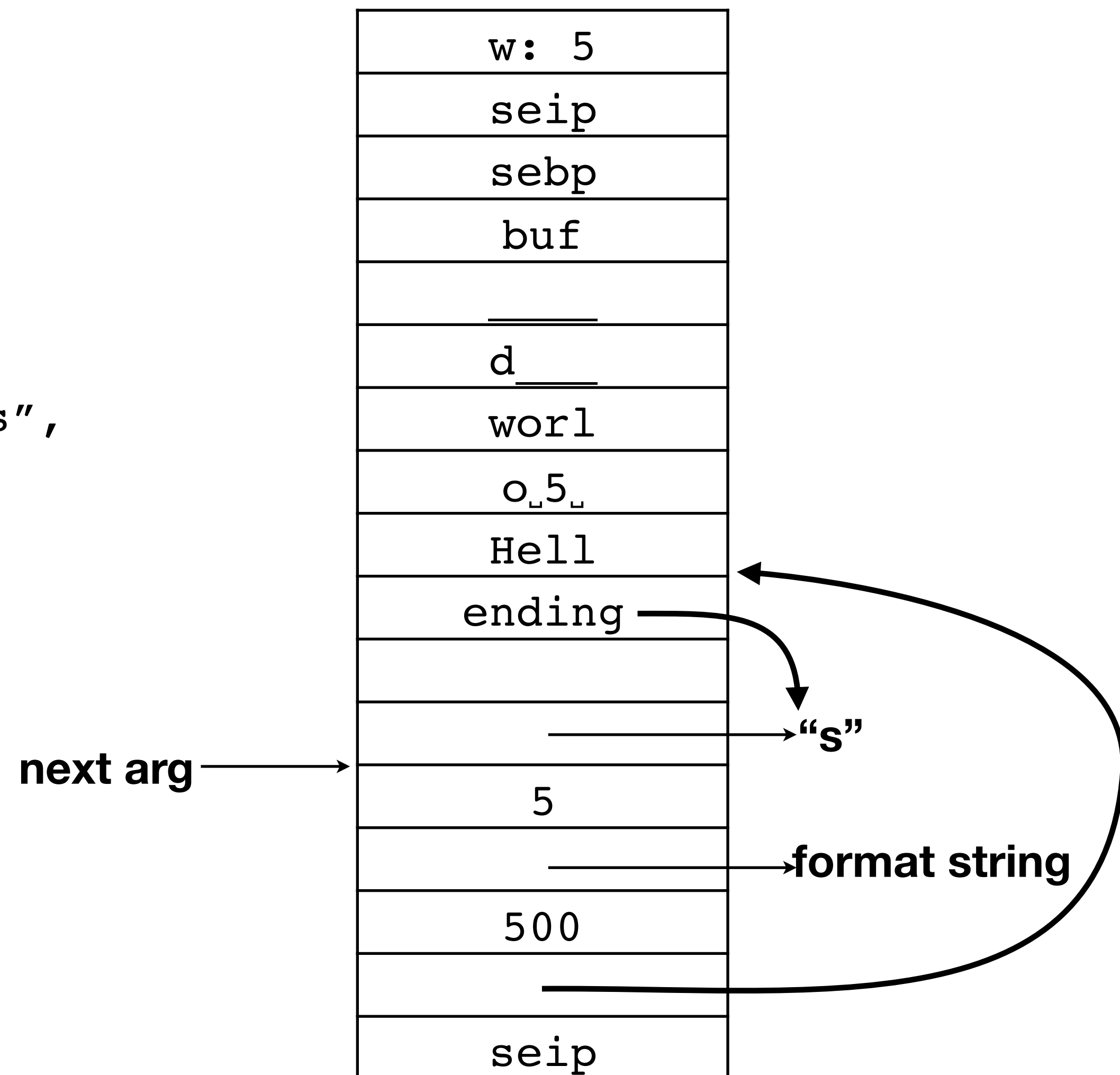
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



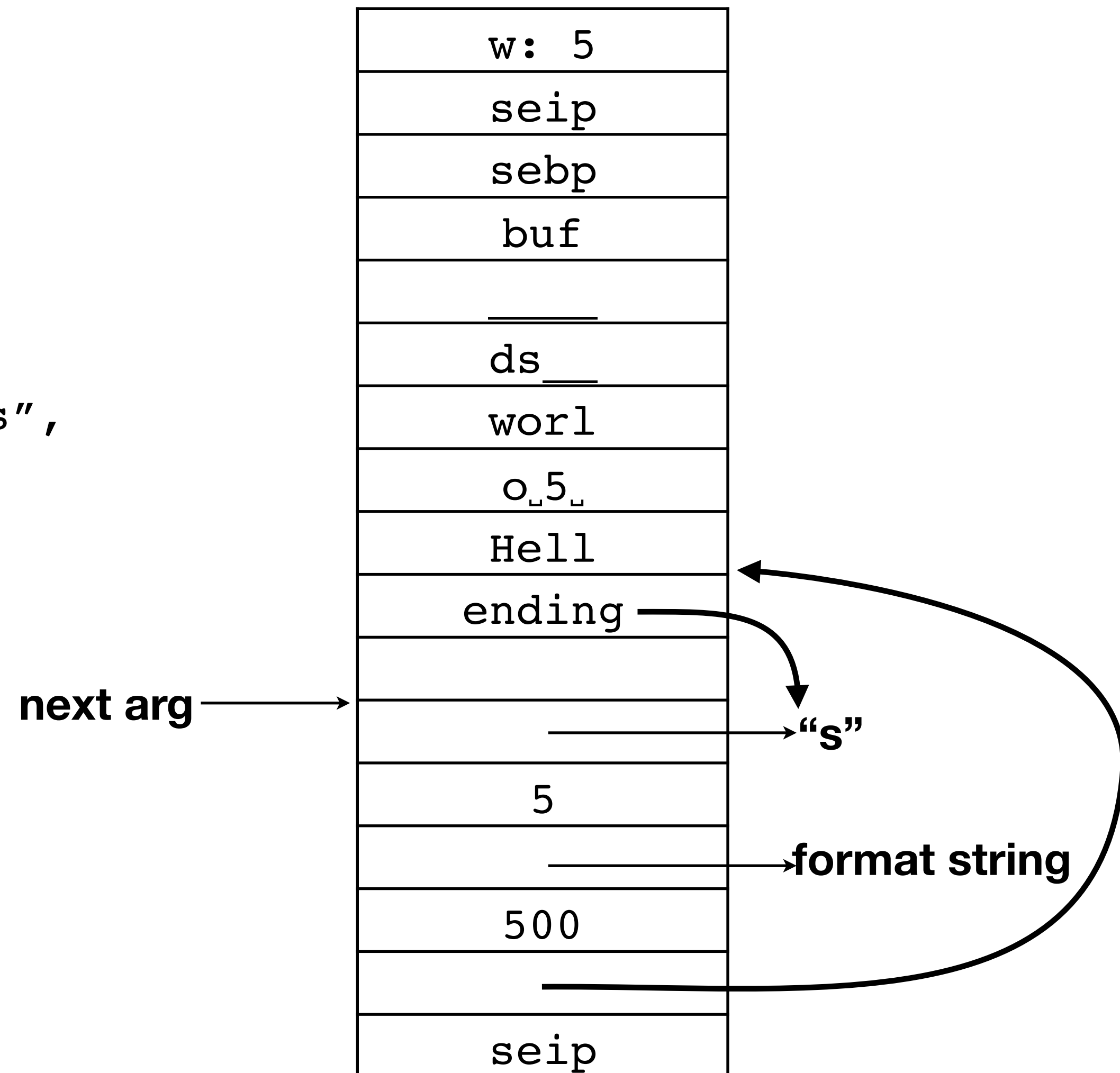
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



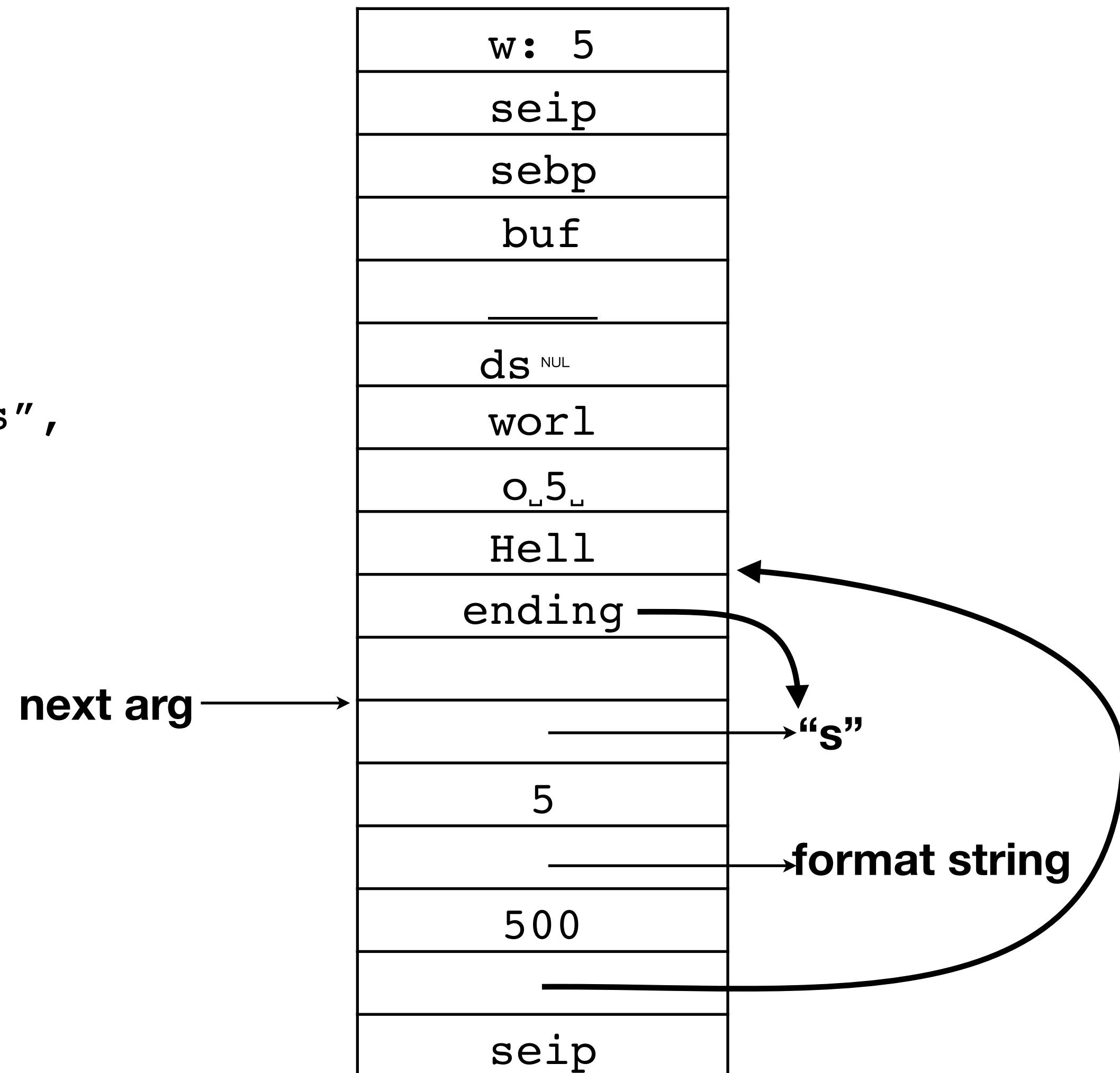
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



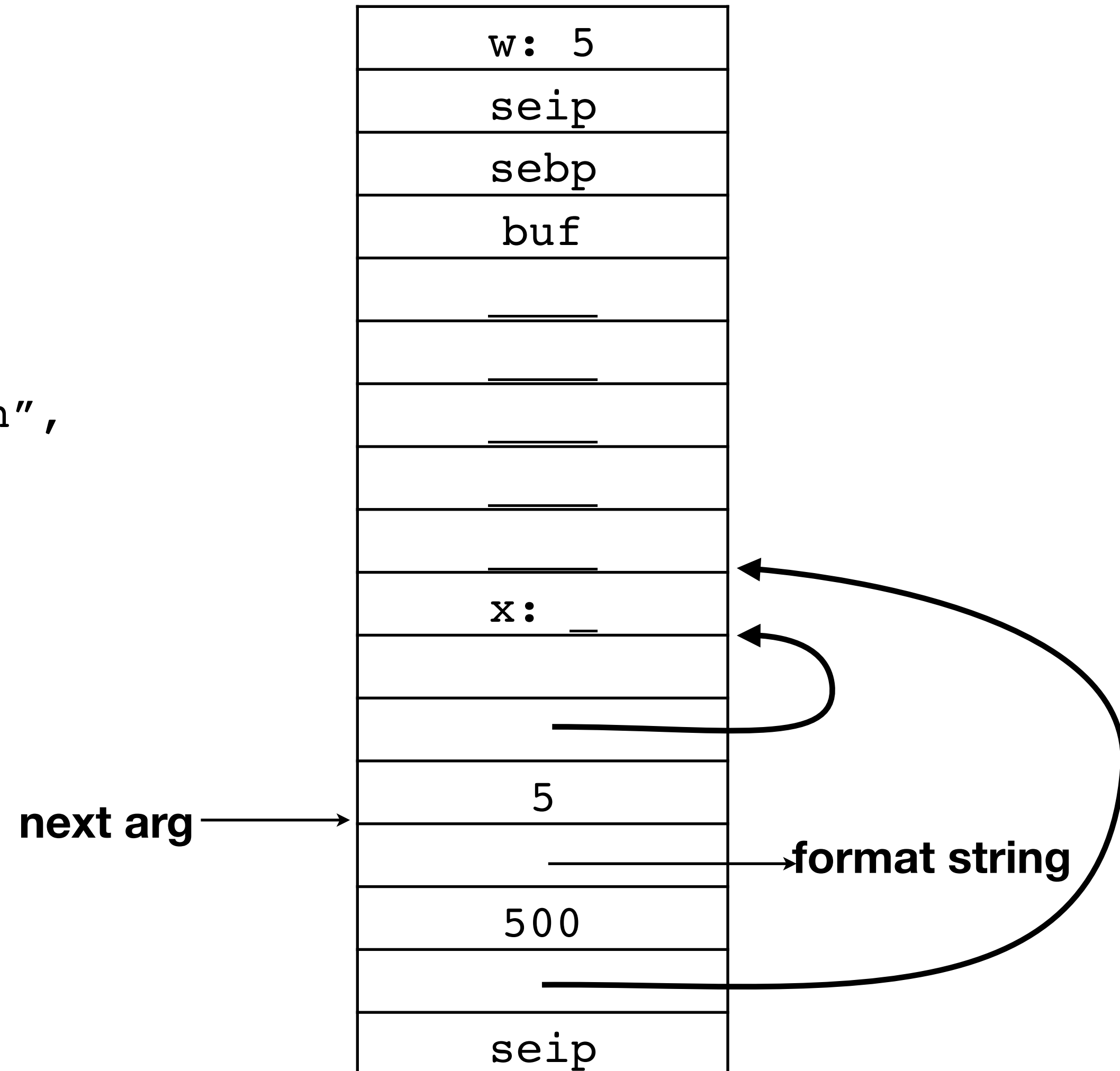
The way snprintf() normally works

```
void foo(int w) {  
    char buf[500];  
    const char *ending = w==1? "" : "s";  
    snprintf(buf, 500, "Hello %d world%s",  
             w, ending);  
}  
...  
foo(5);
```



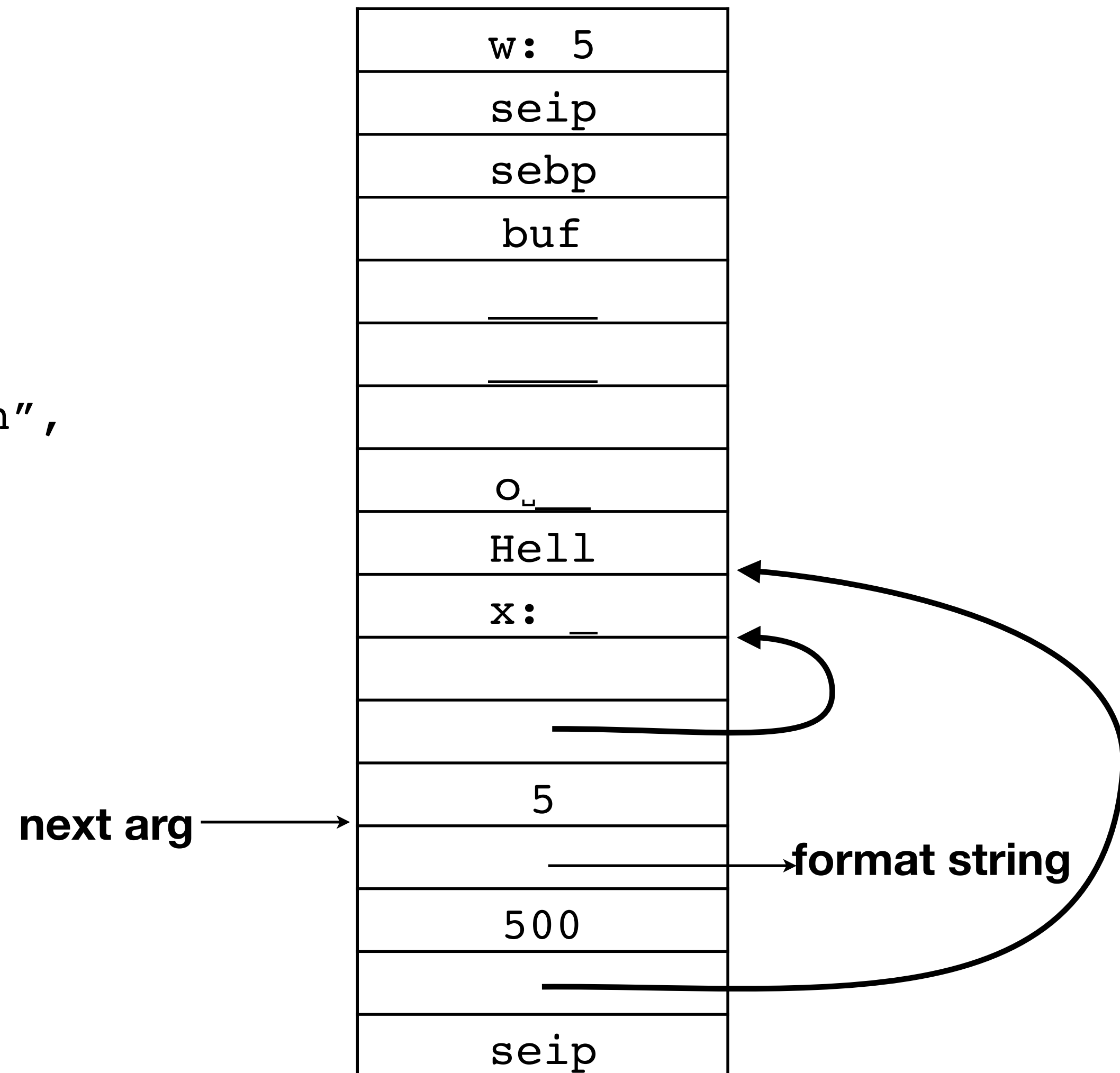
Now with %n

```
void foo(int w) {  
    char buf[500];  
    int x;  
    snprintf(buf, 500, "Hello %d world%n",  
             w, &x);  
}  
...  
foo(5);
```



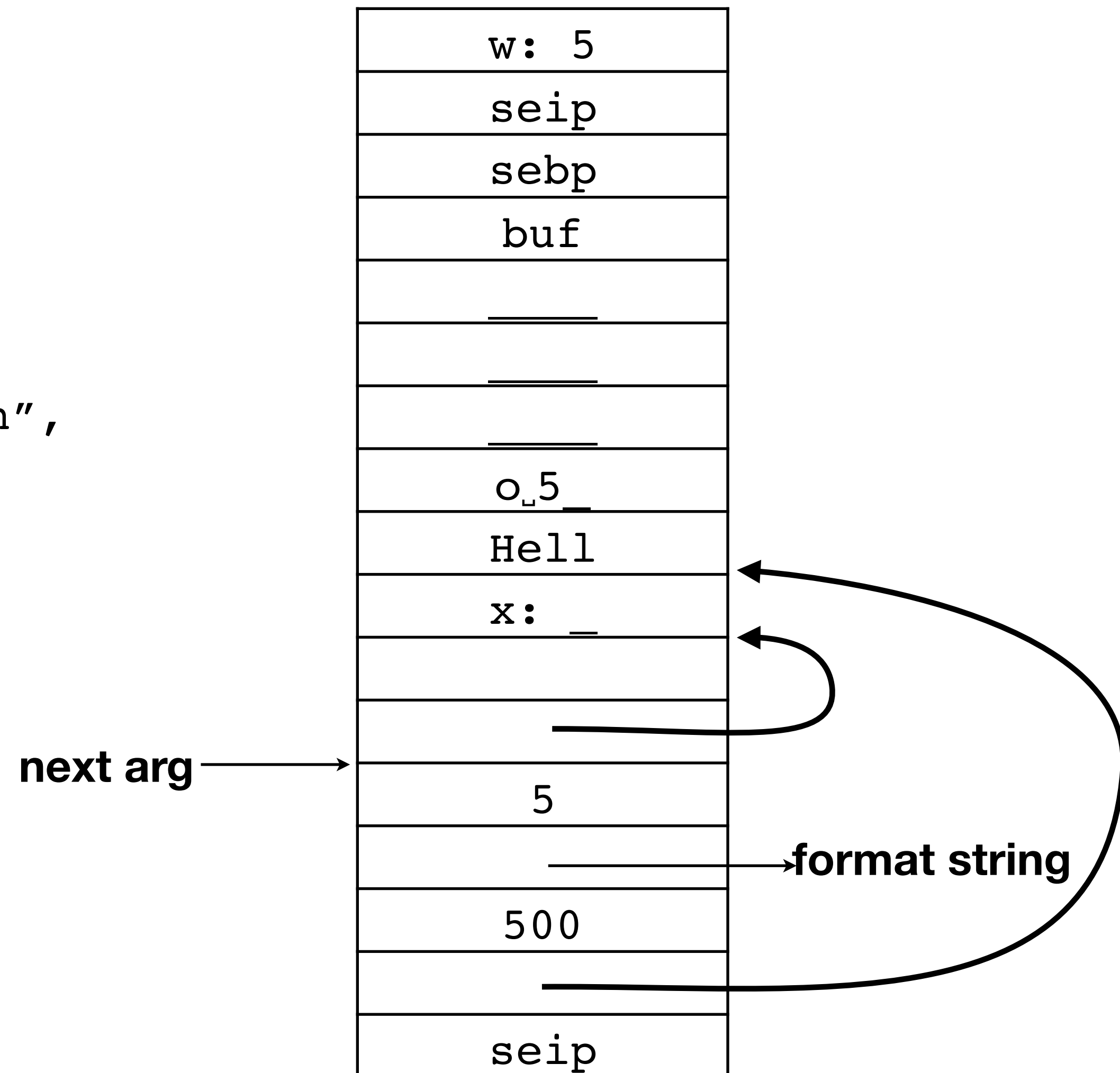
Now with %n

```
void foo(int w) {  
    char buf[500];  
    int x;  
    snprintf(buf, 500, "Hello %d world%n",  
              w, &x);  
}  
...  
foo(5);
```



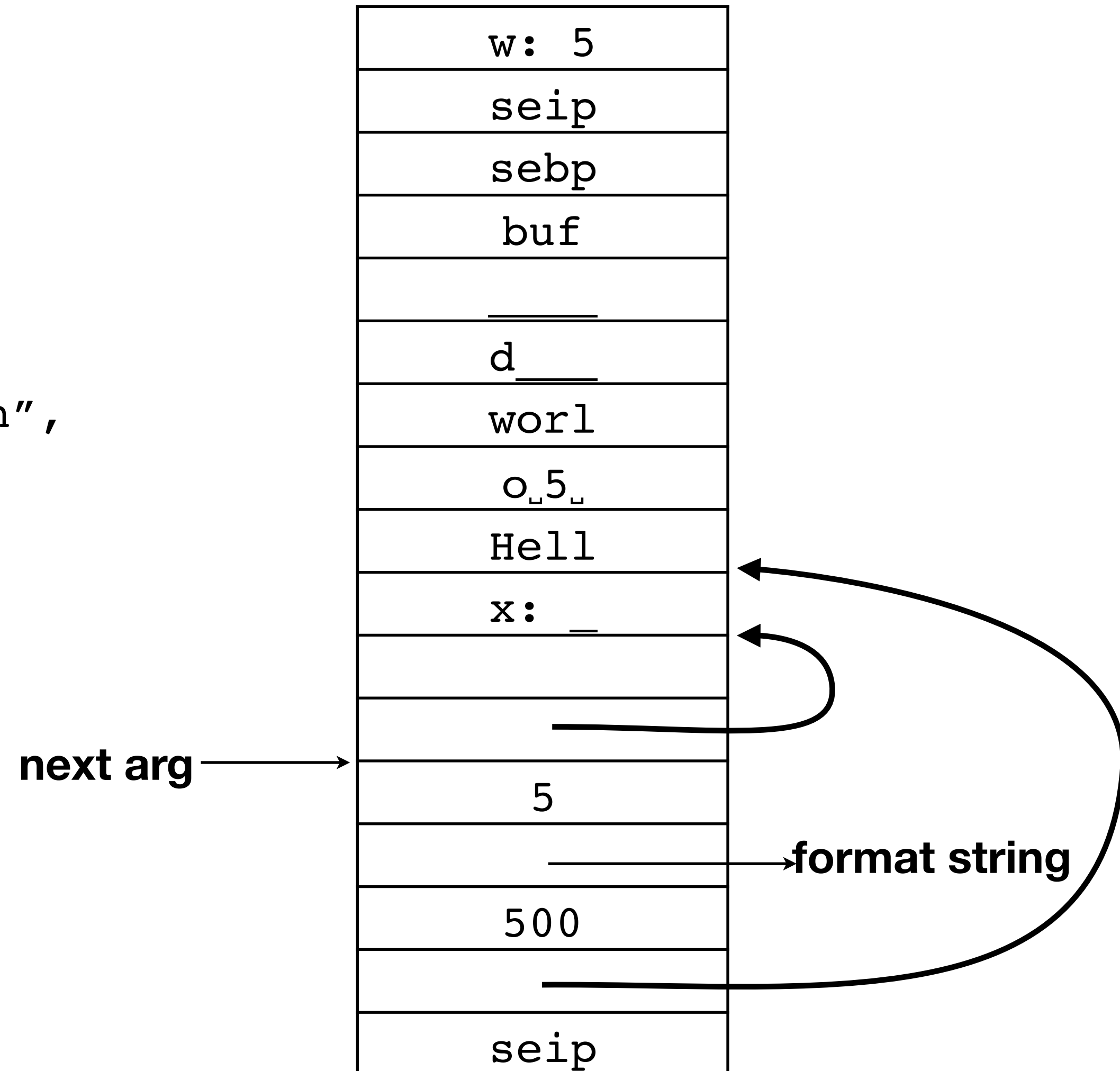
Now with %n

```
void foo(int w) {  
    char buf[500];  
    int x;  
    snprintf(buf, 500, "Hello %d world%n",  
             w, &x);  
}  
...  
foo(5);
```



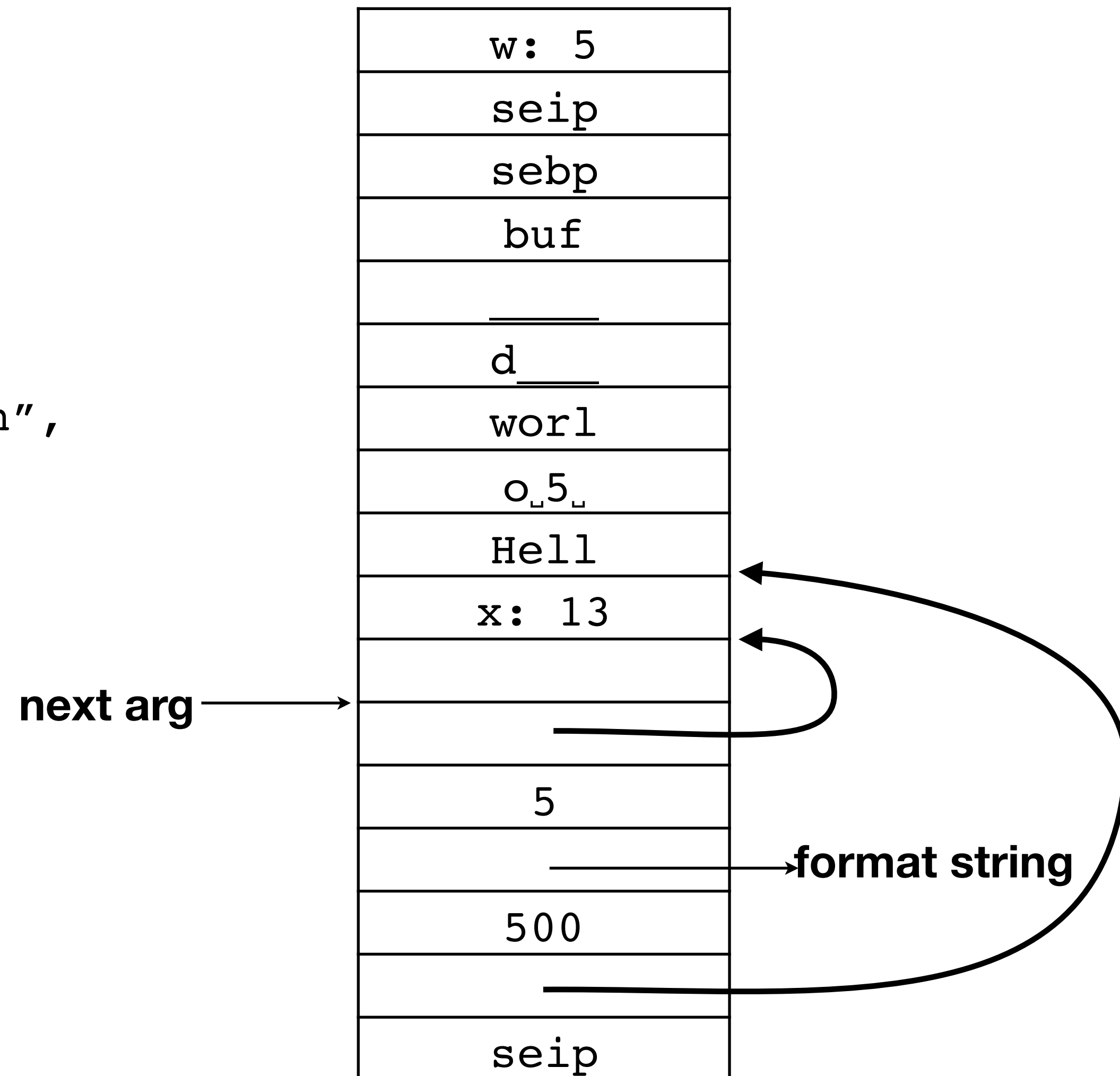
Now with %n

```
void foo(int w) {  
    char buf[500];  
    int x;  
    snprintf(buf, 500, "Hello %d world%n",  
             w, &x);  
}  
...  
foo(5);
```



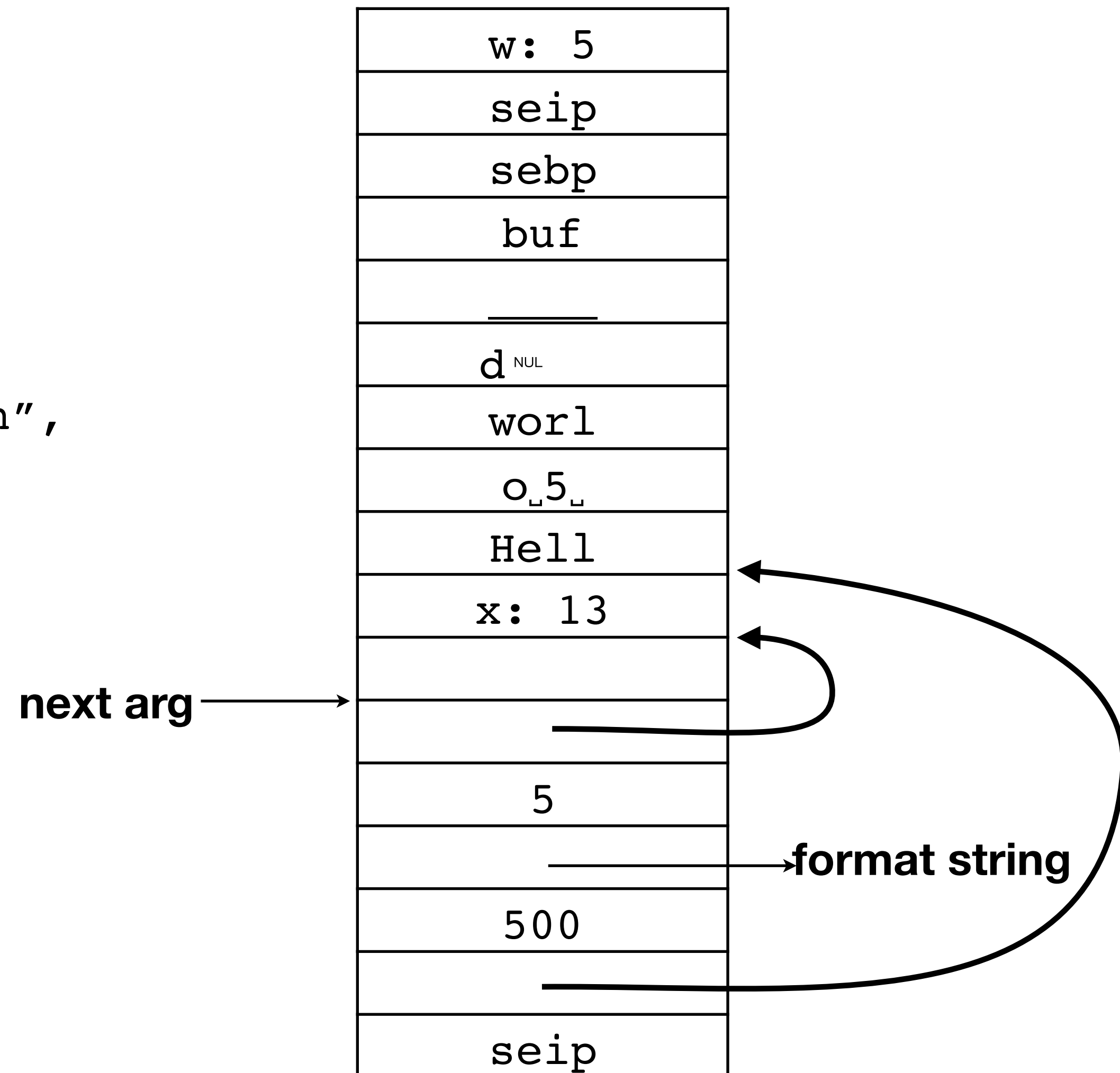
Now with %n

```
void foo(int w) {
    char buf[500];
    int x;
    snprintf(buf, 500, "Hello %d world%n",
              w, &x);
}
...
foo(5);
```



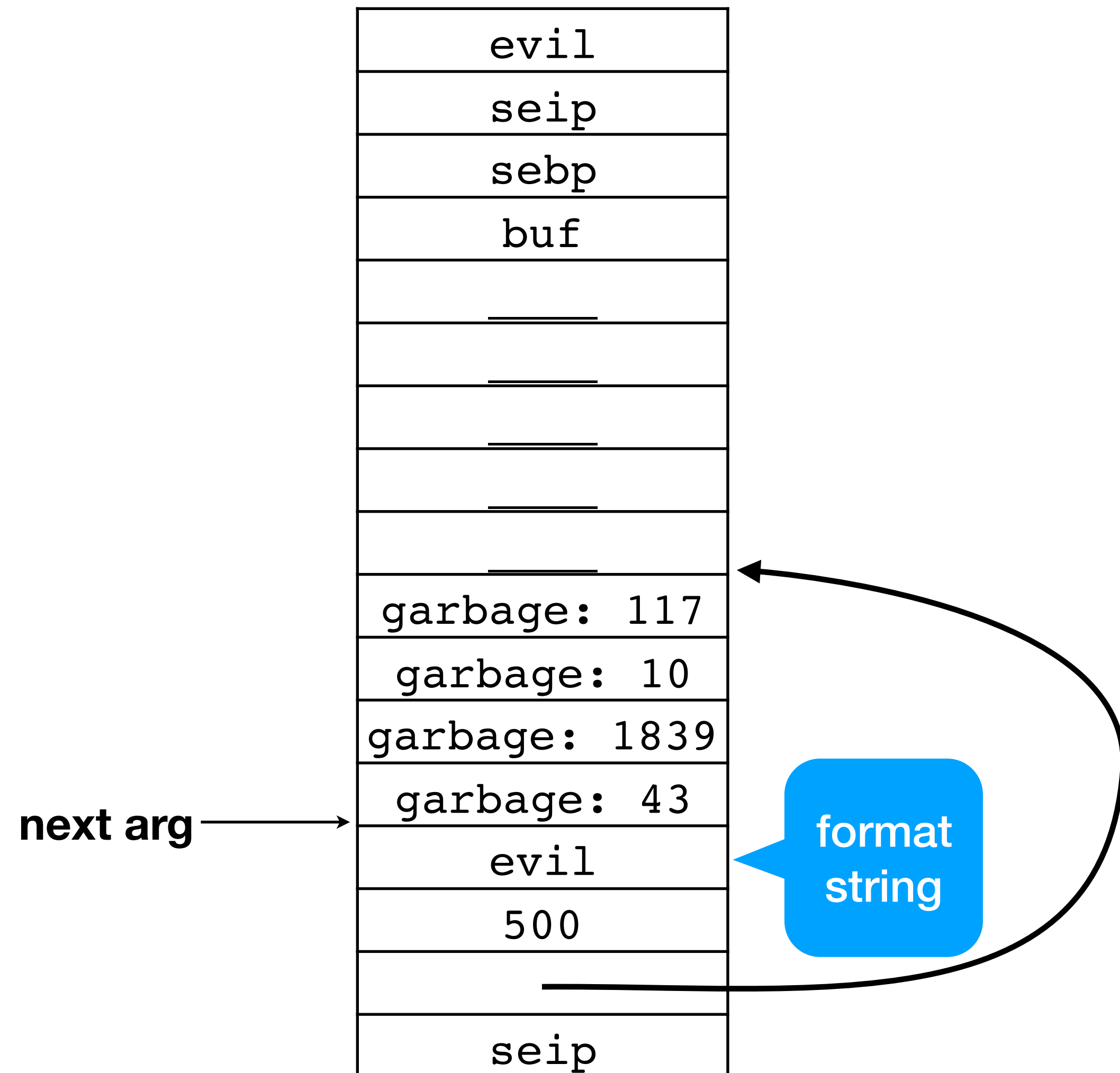
Now with %n

```
void foo(int w) {  
    char buf[500];  
    int x;  
    snprintf(buf, 500, "Hello %d world%n",  
             w, &x);  
}  
...  
foo(5);
```



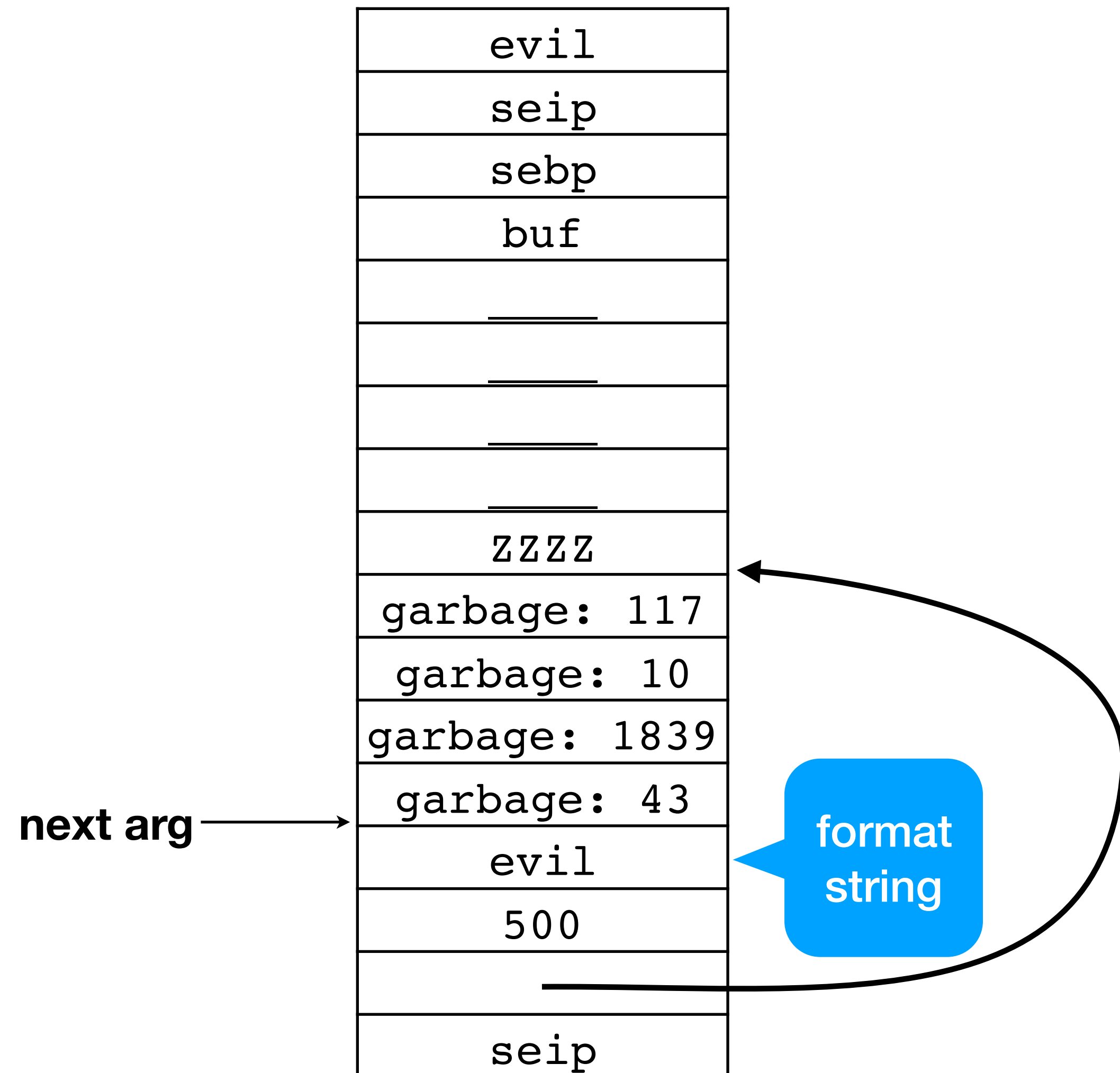
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "ZZZZ%x%x%x%x%x" );
```



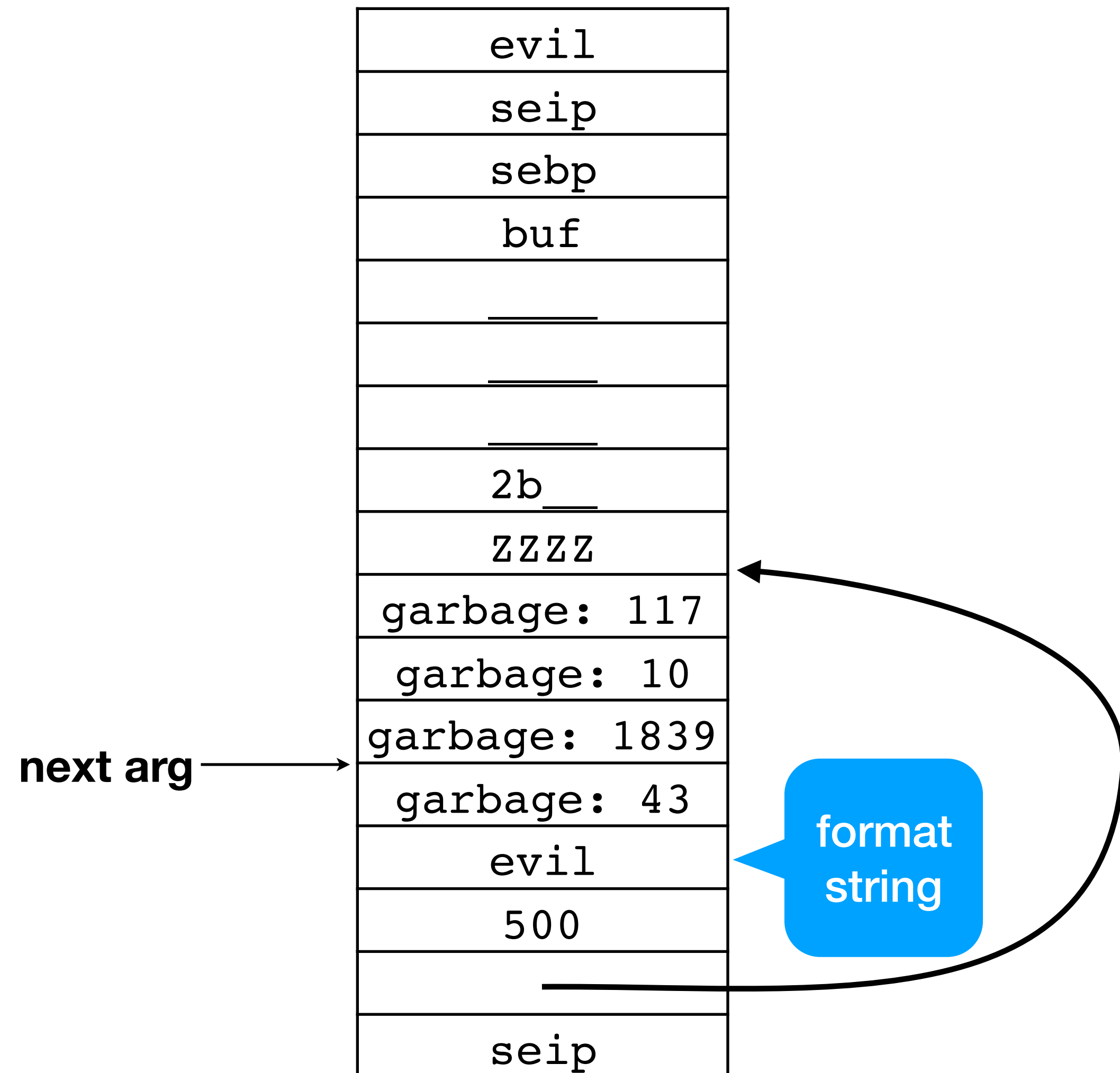
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "ZZZZ%x%x%x%x%x" );
```



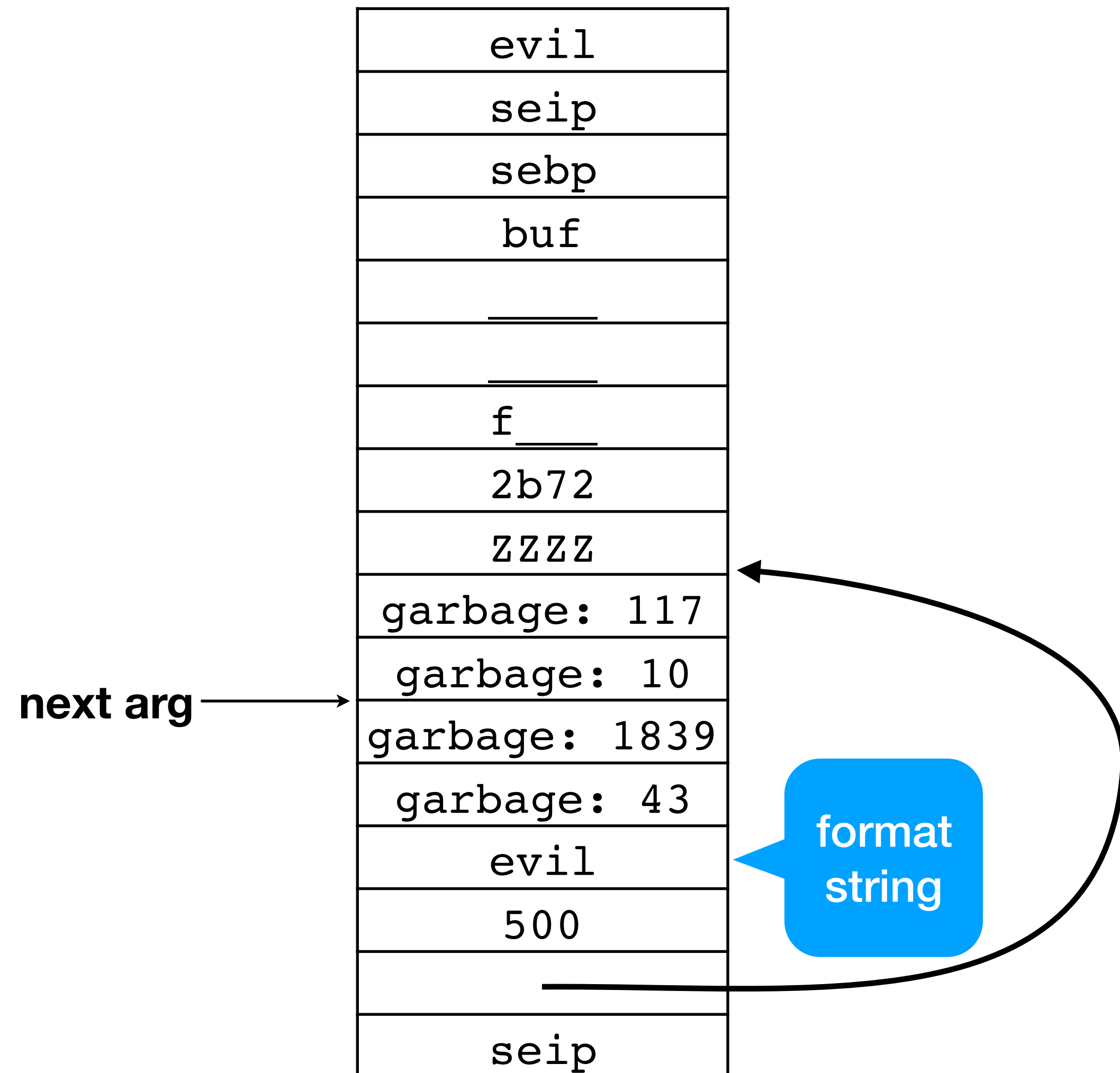
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo("ZZZZ%x%x%x%x%x");
```



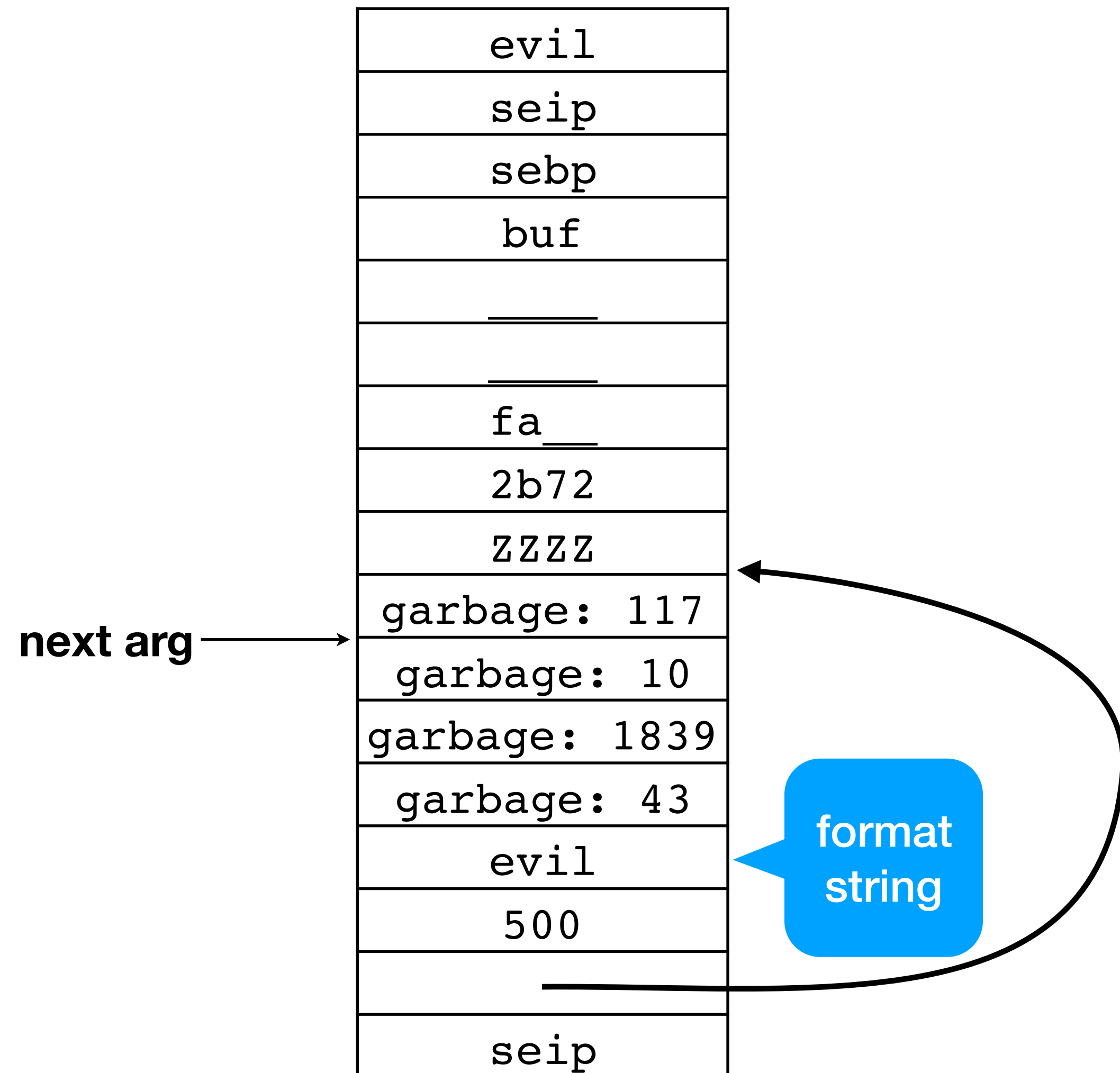
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo("ZZZZ%x%x%x%x%x");
```



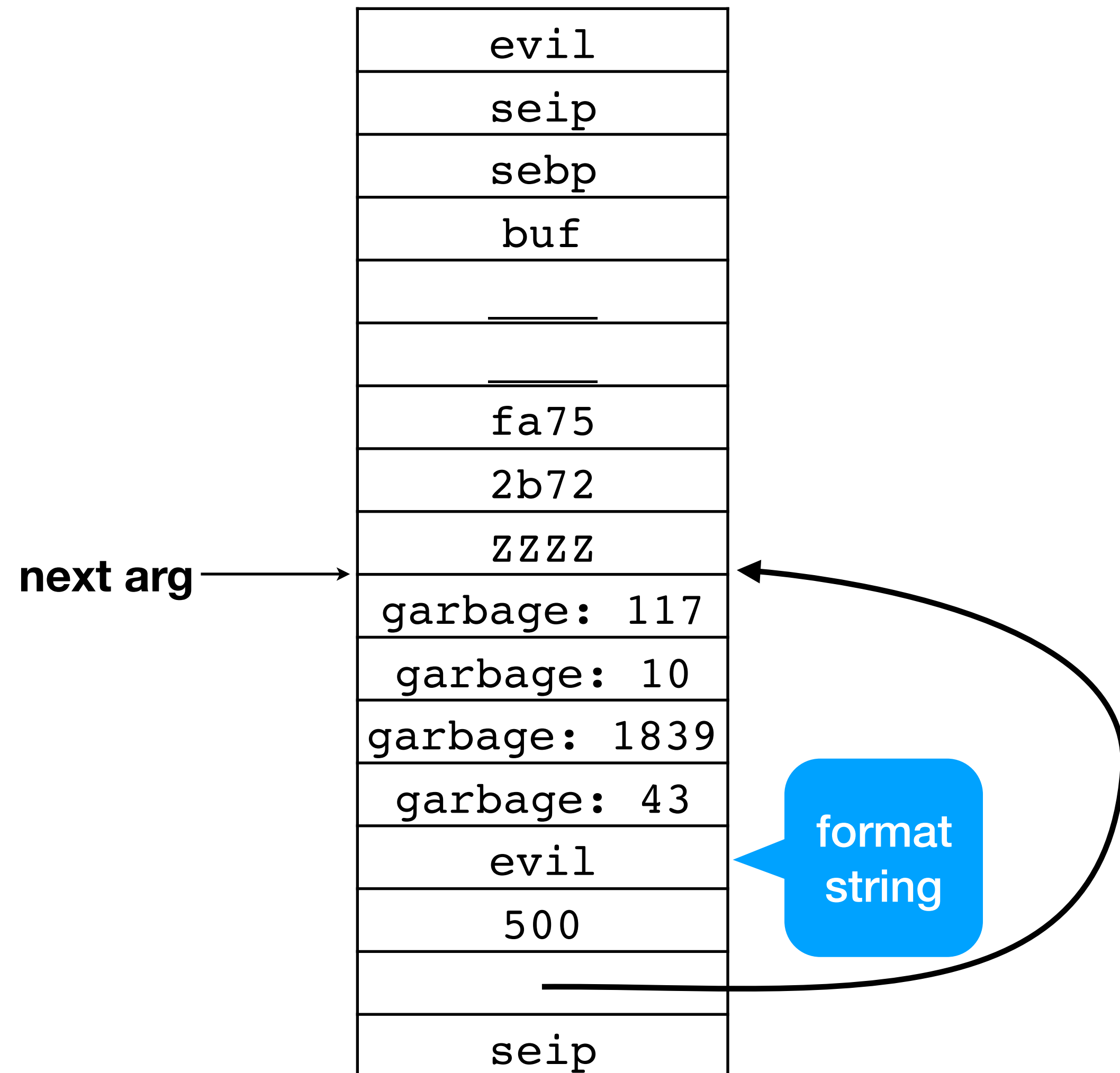
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo("ZZZZ%x%x%x%x%x");
```



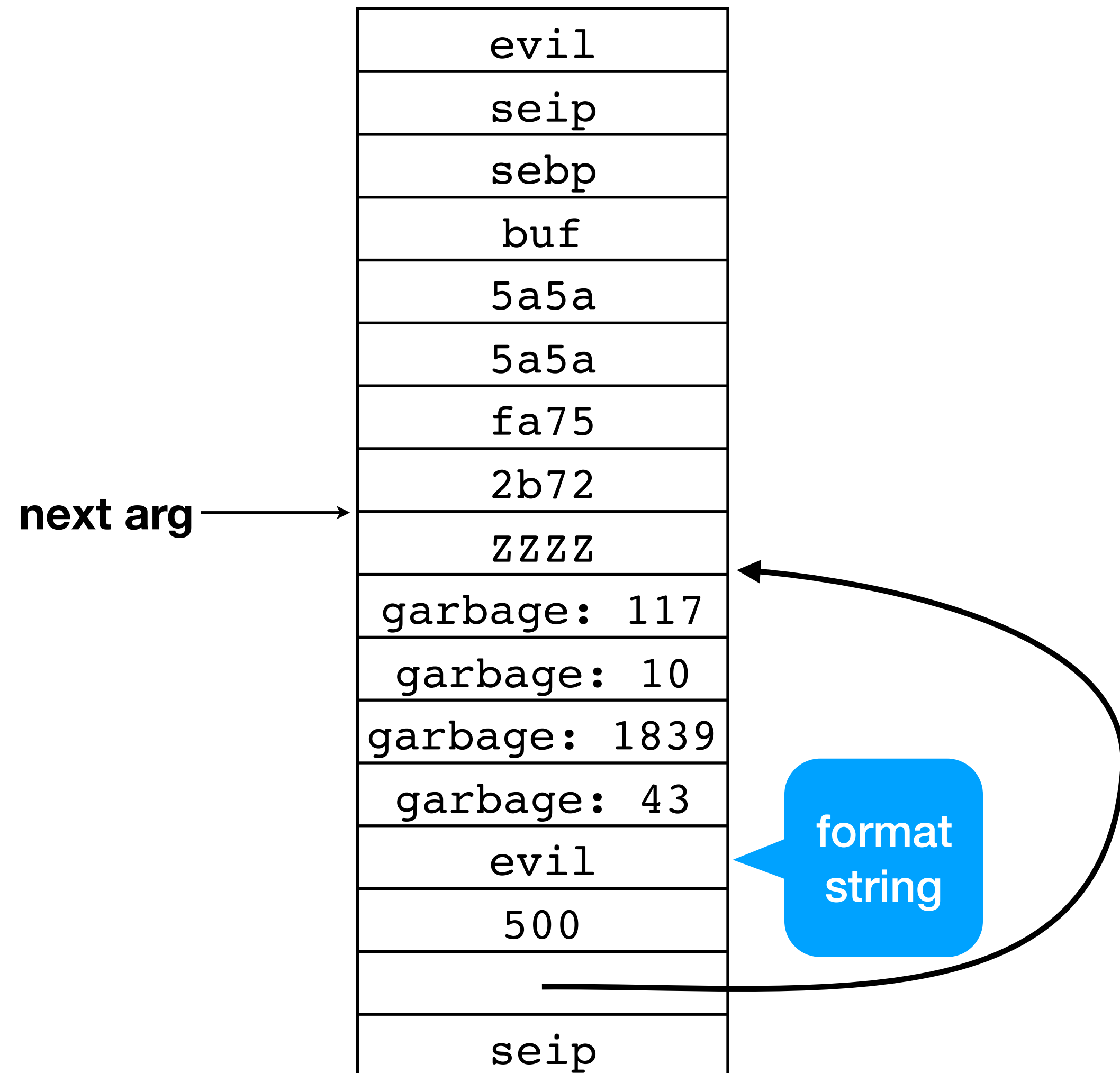
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "ZZZZ%x%x%x%x%x" );
```



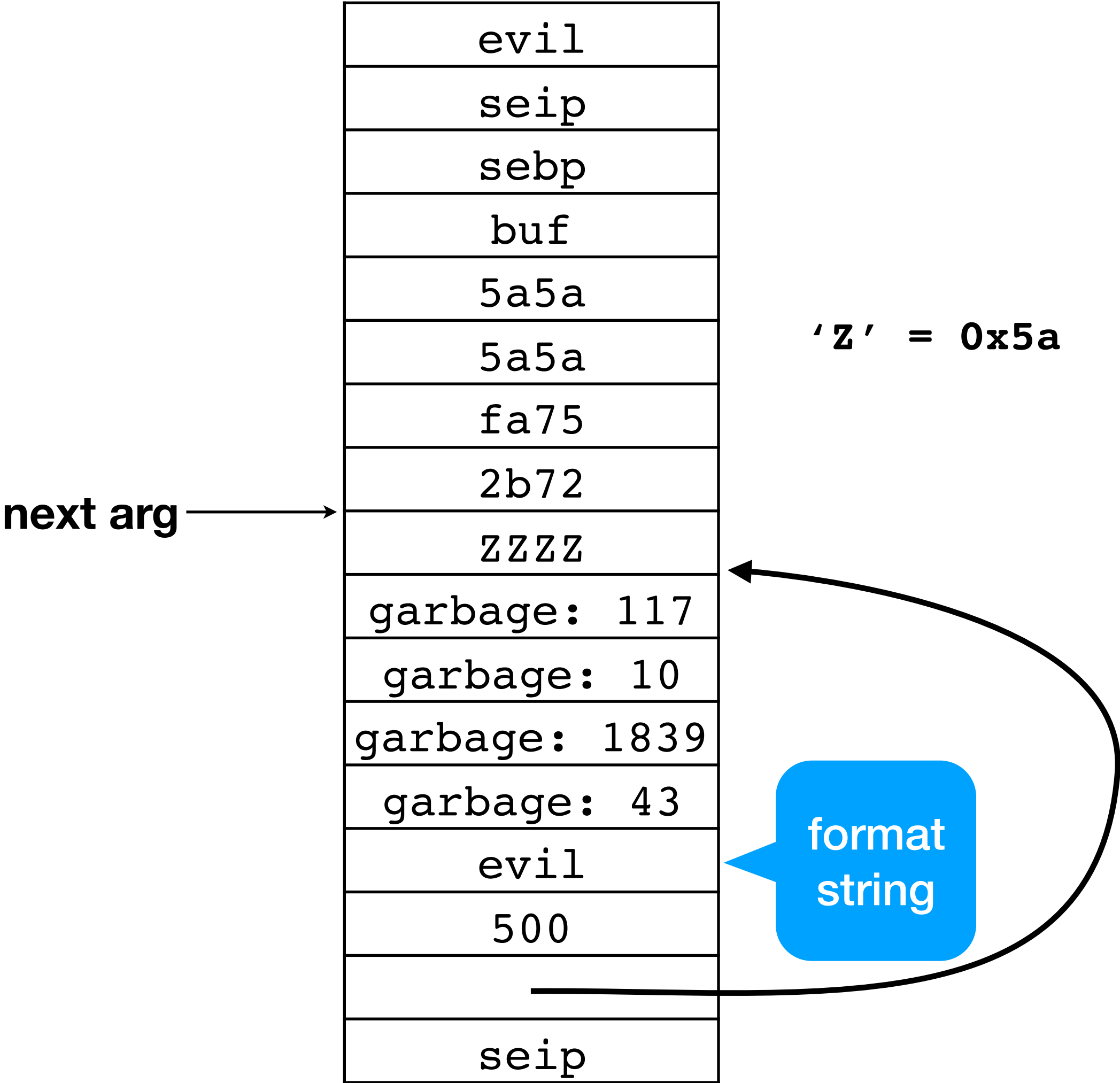
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo("ZZZZ%x%x%x%x%x");
```



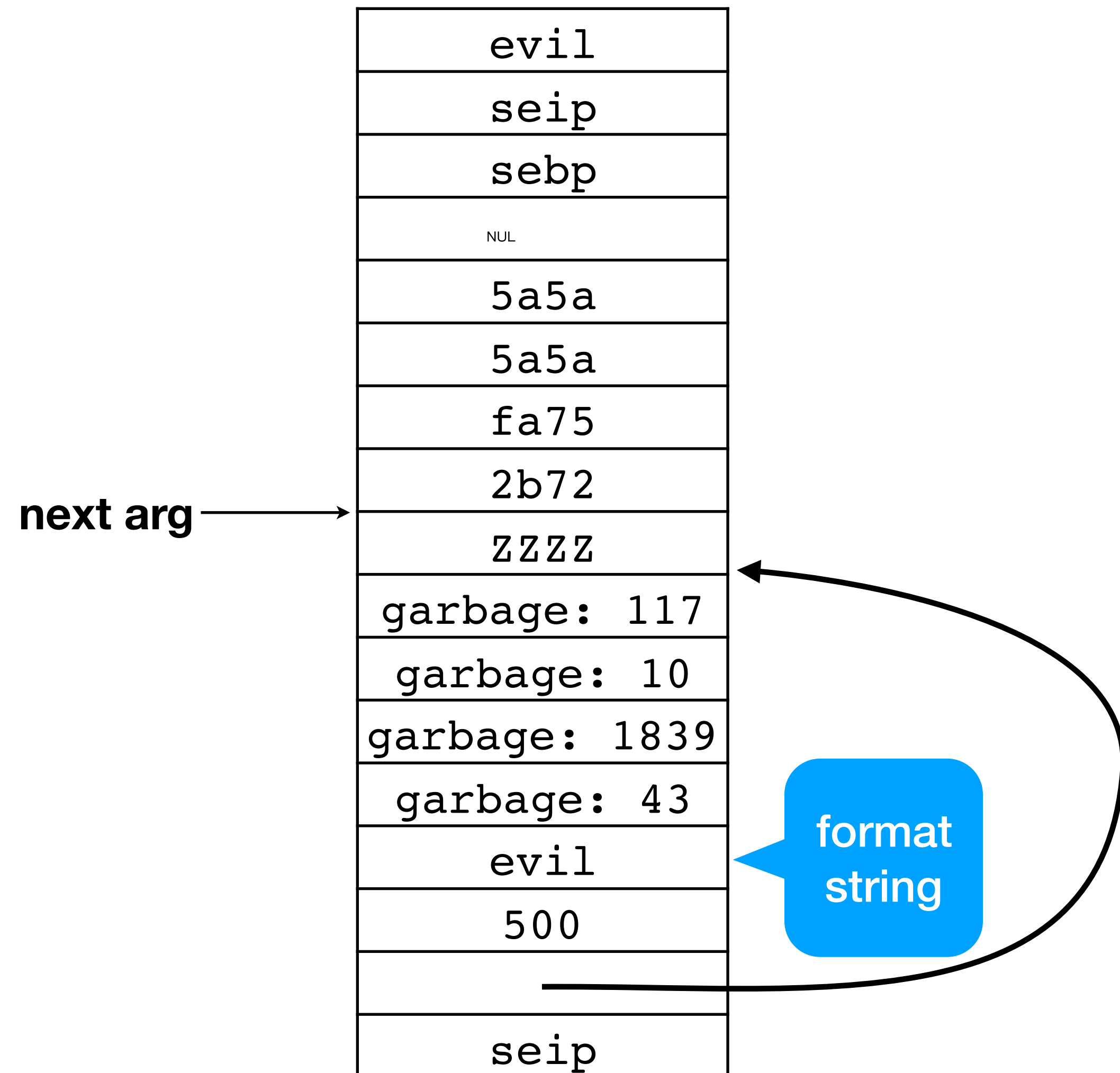
Attacker controlled format string

```
void foo(const char *evil) {
    char buf[500];
    snprintf(buf, 500, evil);
}
...
foo( "ZZZZ%x%x%x%x%x" );
```



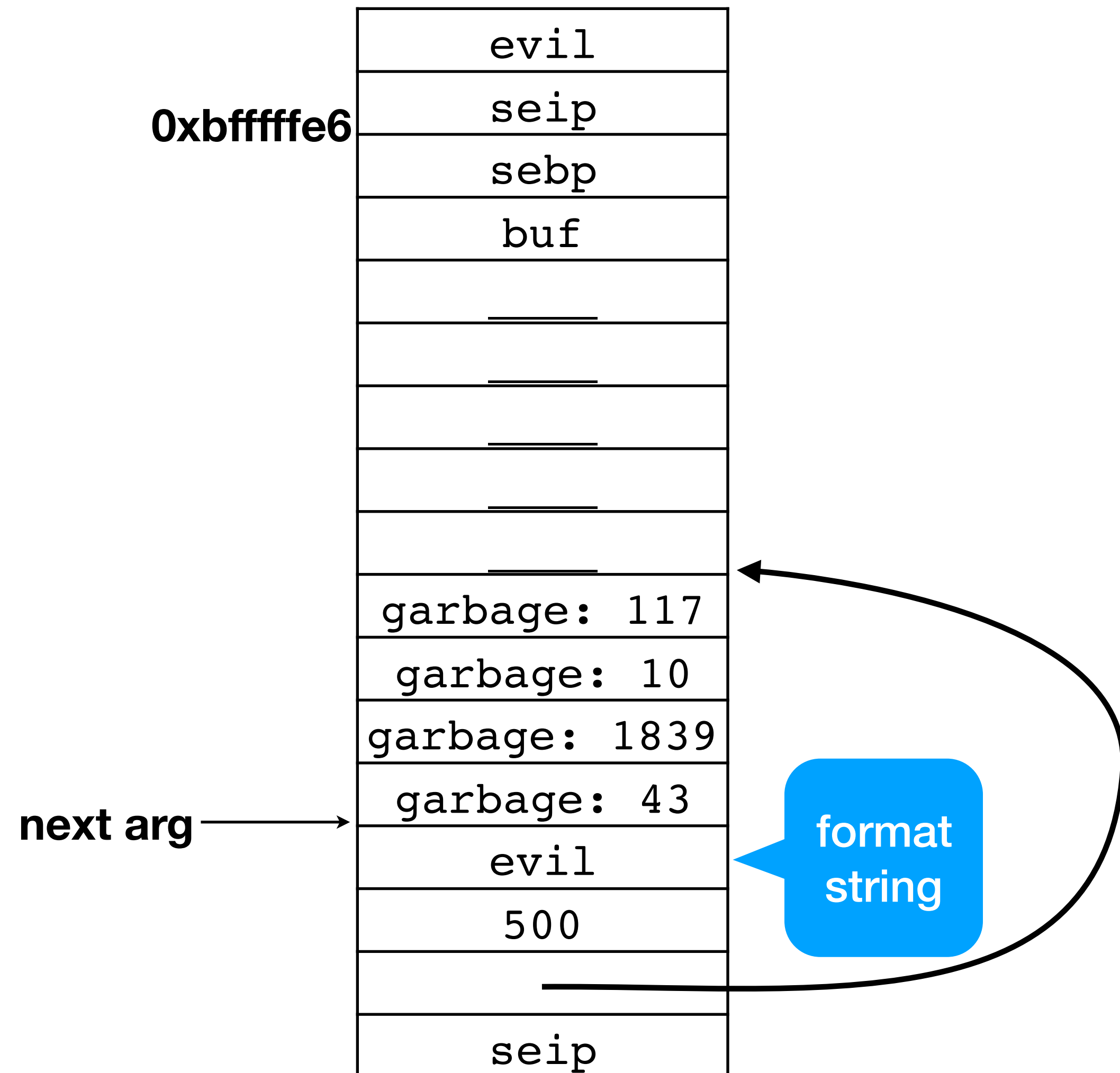
Attacker controlled format string

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "ZZZZ%x%x%x%x%x" );
```



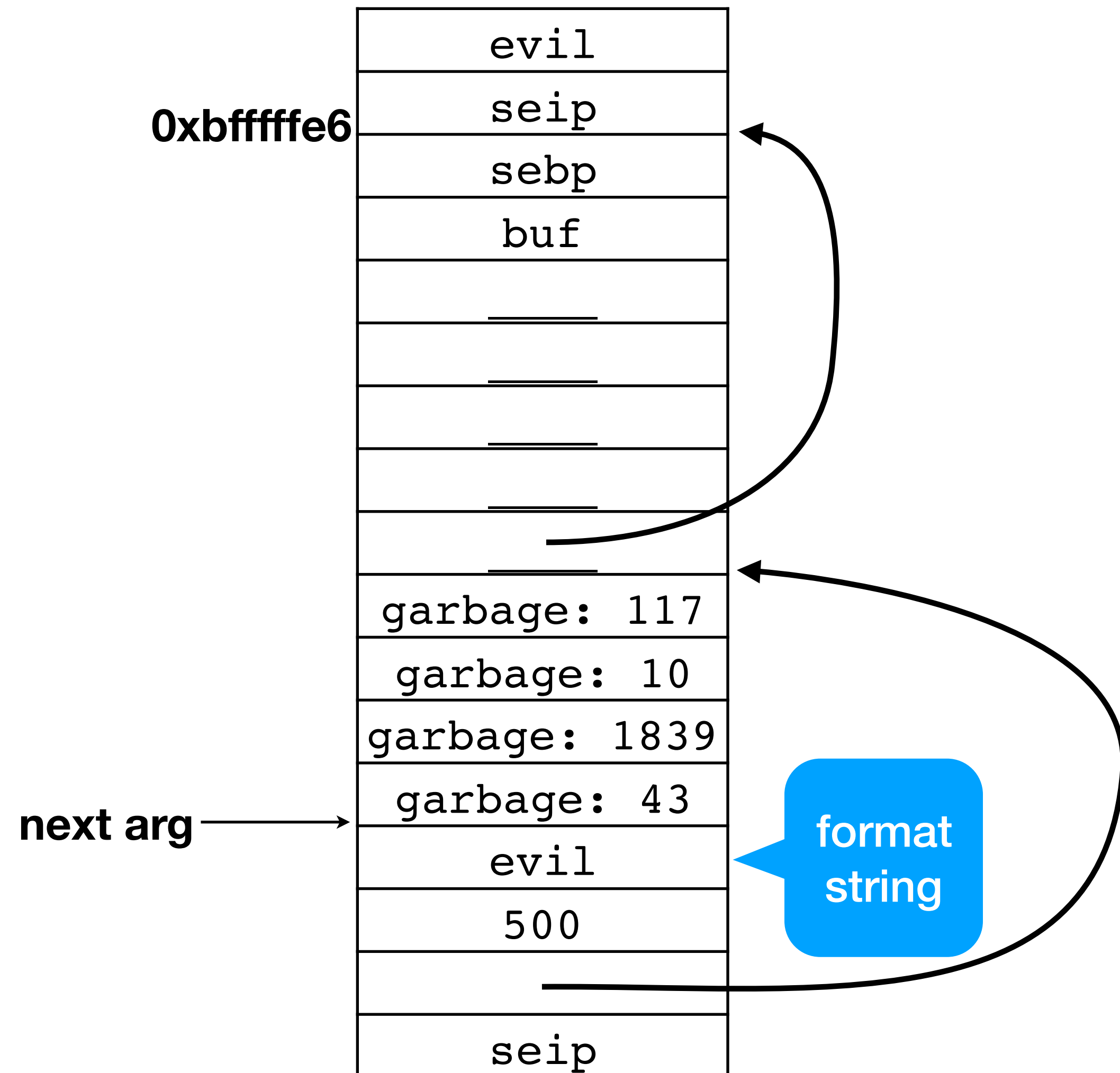
Overwriting seip

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "\xe6\xff\xff\xbf%x%x%x%x%n" );
```



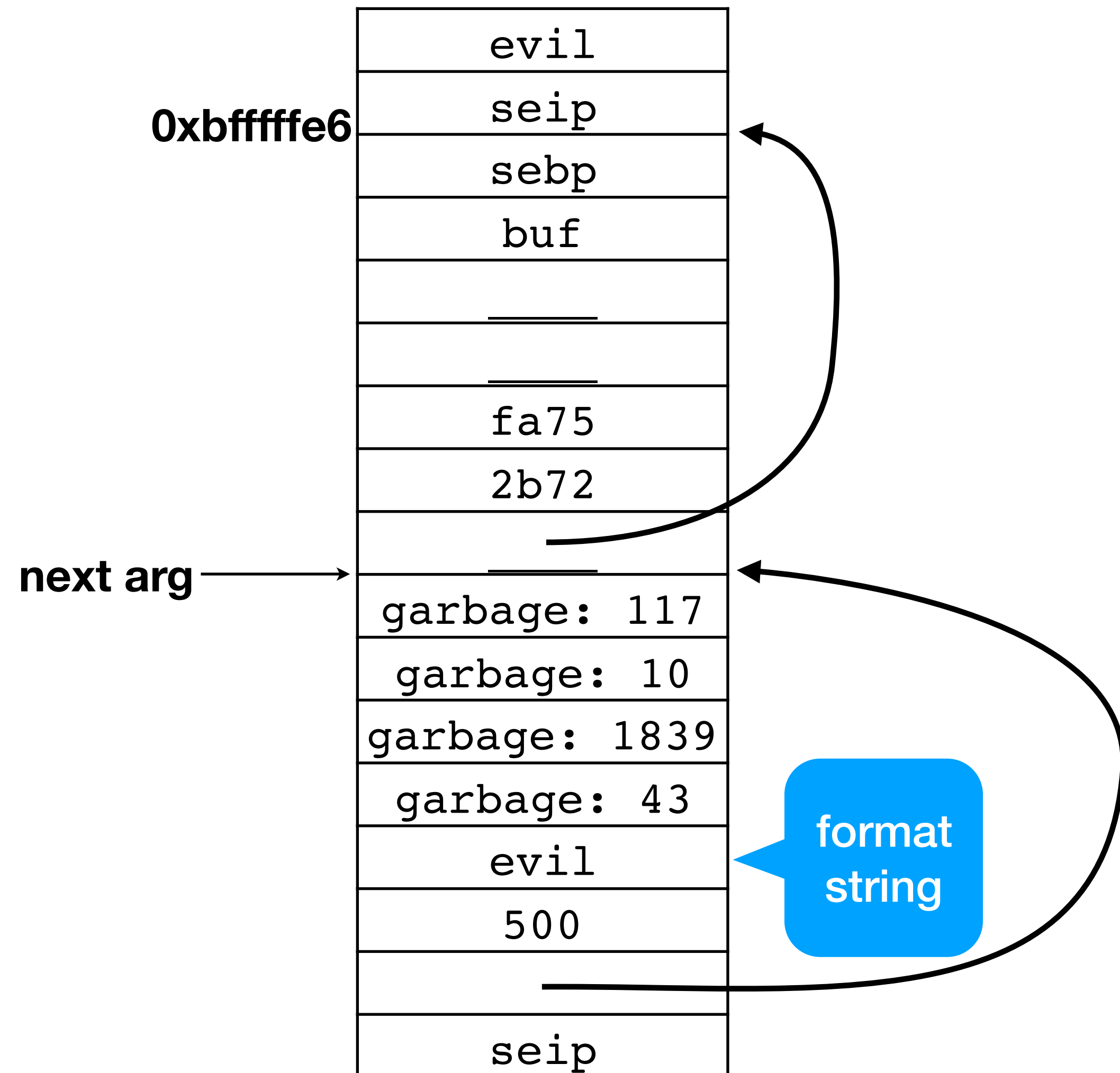
Overwriting seip

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "\xe6\xff\xff\xbf%x%x%x%x%n" );
```



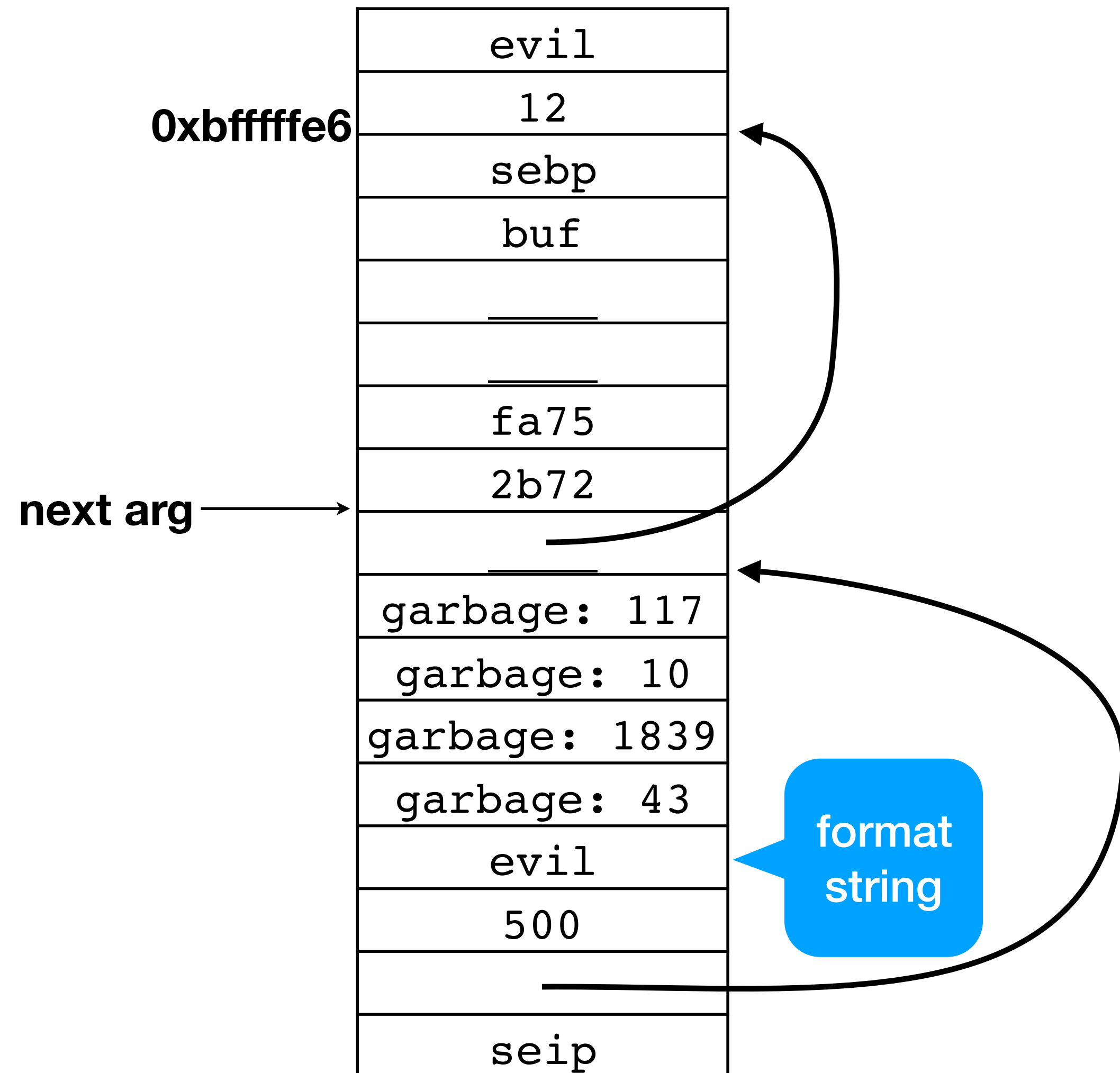
Overwriting seip

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "\xe6\xff\xff\xbf%x%x%x%x%n" );
```



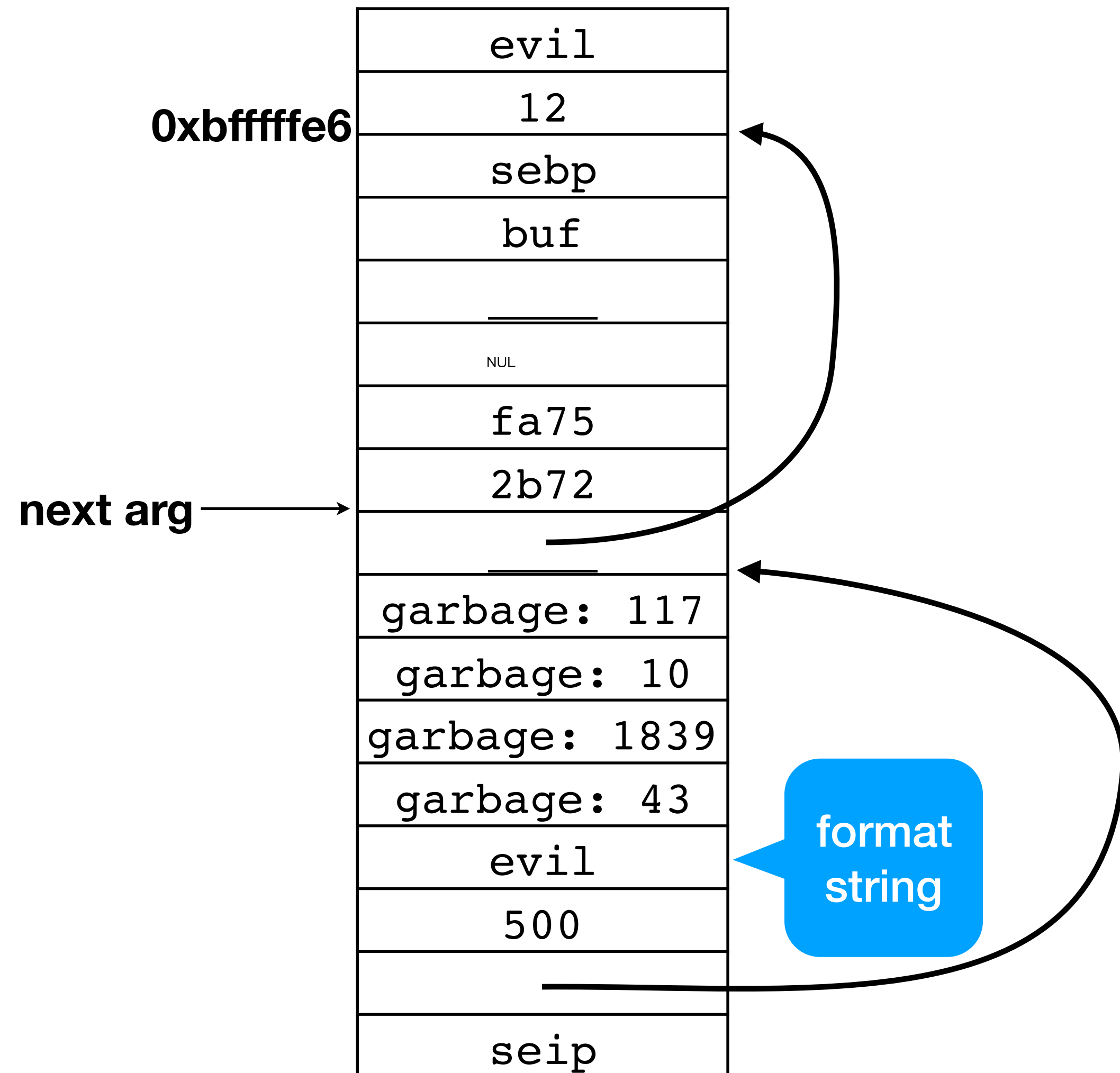
Overwriting seip

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "\xe6\xff\xff\xbf%x%x%x%x%n" );
```



Overwriting seip

```
void foo(const char *evil) {  
    char buf[500];  
    snprintf(buf, 500, evil);  
}  
...  
foo( "\xe6\xff\xff\xbf%x%x%x%x%n" );
```



Picking the bytes to write

- Use `%<len>x` to control the length of the output
- Use `%hhn` to write just the least-significant byte of the length

Almost putting it all together

```
evil = "<address>ZZZZ"  
      "<address+1>ZZZZ"  
      "<address+2>ZZZZ"  
      "<address+3>"  
      "%8x%8x...%8x"  
      "%<len>x%hhn"  
      "%<len>x%hhn"  
      "%<len>x%hhn"  
      "%<len>x%hhn";
```

Misaligned buf

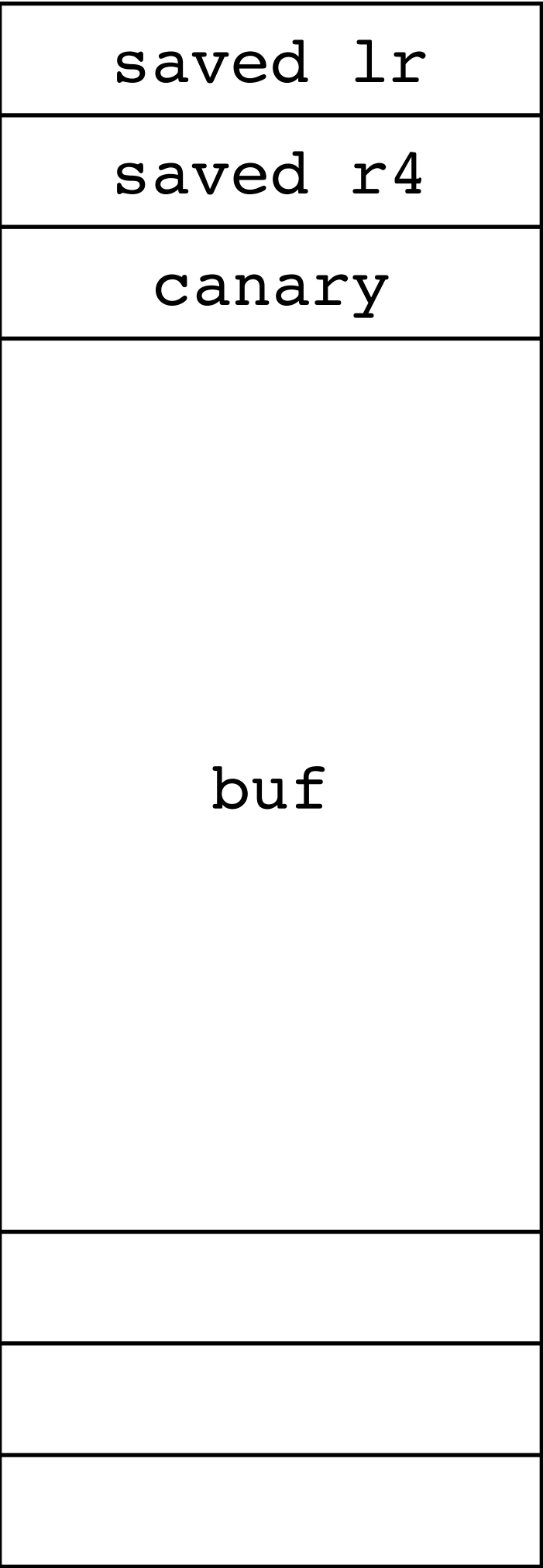
- If `buf` is not 4-byte aligned, prepend 1, 2, or 3 characters to `evil`

Advantages of format string exploits

- No need to smash the stack (targeted write)
- Avoids defenses such as stack canaries!
 - Stack canary is a random word pushed onto the stack that is checked before the function returns

Stack Canaries

```
int bar(char *);
char foo(void) {
    char buf[100];
    bar(buf);
    return buf[0];
}
foo:
    push    {r4, lr}
    sub     sp, sp, #104
    movw    r4, #:lower16:__stack_chk_guard
    movt    r4, #:upper16:__stack_chk_guard
    ldr     r3, [r4]
    str     r3, [sp, #100]
    mov     r0, sp
    bl      bar
    ldrb    r0, [sp]           @ zero_extendqisi2
    ldr     r2, [sp, #100]
    ldr     r3, [r4]
    cmp     r2, r3
    beq     .L2
    bl      __stack_chk_fail
.L2:
    add     sp, sp, #104
    pop     {r4, pc}
```



Disadvantages of format string exploits

- Easy to catch so rarer:

```
$ gcc -Wformat=2 f.c
```

```
f.c: In function 'main':
```

```
f.c:5: warning: format not a string literal and no format arguments
```

- Tricky to exploit compared to buffer overflows

What else can we overwrite?

- Function pointers
- C++ vtables
- Global offset table (GOT)

Function pointers

```
#include <stdlib.h>
#include <stdio.h>

int compare(const void *a,
            const void *b) {
    const int *x = a;
    const int *y = b;
    return *x - *y;
}

int main() {
    int i;
    int arr[6] = {2, 1, 5, 13, 8, 4};
    qsort(arr, 6, 4, compare);
    for (i = 0; i < 6; ++i)
        printf("%d ", arr[i]);
    putchar('\n');
    return 0;
}
```

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    ...
    leal     24(%esp), %esi // arr
    ...
    movl     $compare, 12(%esp)
    movl     $4, 8(%esp)
    movl     $6, 4(%esp)
    movl     %esi, (%esp)
    call     qsort

qsort:
    ...
    call     *0x14(%ebp)
    ...
```

C++ Virtual function tables (vtable)

```
struct Foo {  
    Foo() { }  
    virtual ~Foo() { }  
    virtual void fun1() { }  
    virtual void fun2() { }  
};
```

```
void bar(Foo &f) {  
    f.fun1();  
    f.fun2();  
}
```

```
int main() {  
    Foo f;  
    foo(f);  
}
```

```
_Z3barR3Foo: // bar(Foo&)  
    pushl    %ebp  
    movl     %esp, %ebp  
    pushl    %ebx  
    subl     $20, %esp  
    movl     8(%ebp), %ebx    // ebx <- f  
    movl     (%ebx), %eax    // eax <- vtable  
    movl     %ebx, (%esp)    // (esp) <- this  
    call     *8(%eax)        // call virtual function  
    movl     (%ebx), %eax    // eax <- vtable  
    movl     %ebx, (%esp)    // (esp) <- this  
    call     *12(%eax)       // call virtual function  
    addl     $20, %esp  
    popl     %ebx  
    popl     %ebp  
    ret
```

vtable for Foo

```
// Real code
_ZN3FooC1Ev:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl $_ZTV3Foo+8, (%eax)
    popl %ebp
    ret
```

```
_ZTV3Foo:
    .long 0
    .long _ZTI3Foo
    .long _ZN3FooD1Ev
    .long _ZN3FooD0Ev
    .long _ZN3Foo4fun1Ev
    .long _ZN3Foo4fun2Ev
```

```
// Demangled
Foo::Foo():
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl vtable for Foo+8, (%eax)
    popl %ebp
    ret
```

address of
vtable+8 stored in
first word of object

```
vtable for Foo:
    .long 0
    .long typeid for Foo
    .long Foo::~~Foo()
    .long Foo::~~Foo()
    .long Foo::fun1()
    .long Foo::fun2()
```

Global Offset Table (GOT)

- Contains pointers to code and data in shared libraries
- Library functions aren't called directly; stub in the Procedure Linkage Table (PLT) called
- E.g., call exit -> call exit@plt
- exit@plt looks up the address of exit in the GOT and jumps to it (not the whole story)
- Overwrite function pointer in GOT