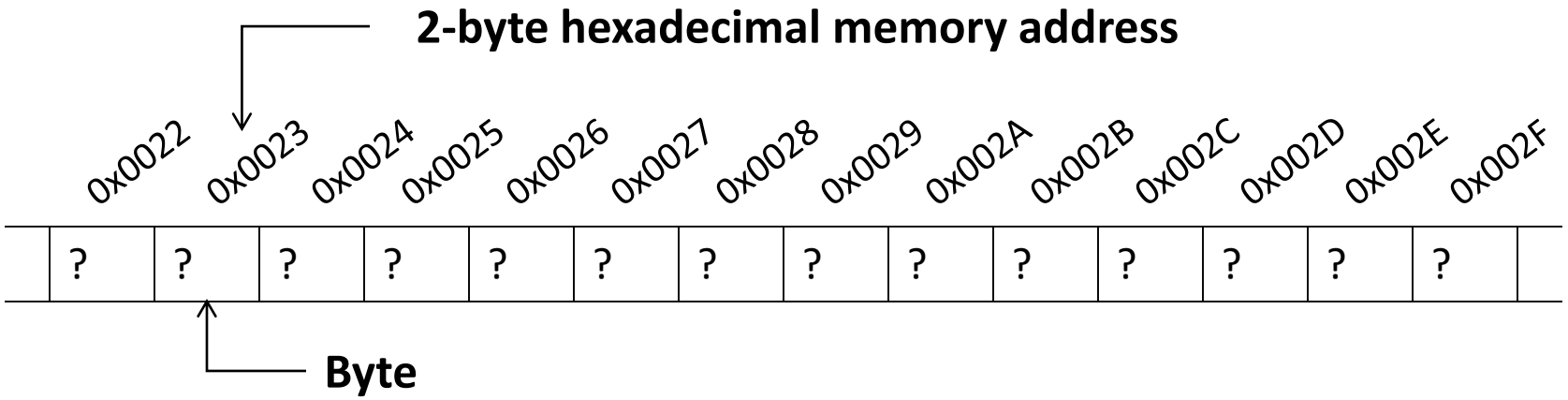
UNIVERSITY of WEST FLORIDA

COP4634: Systems & Networks I

Memory & Strings

Memory Layout

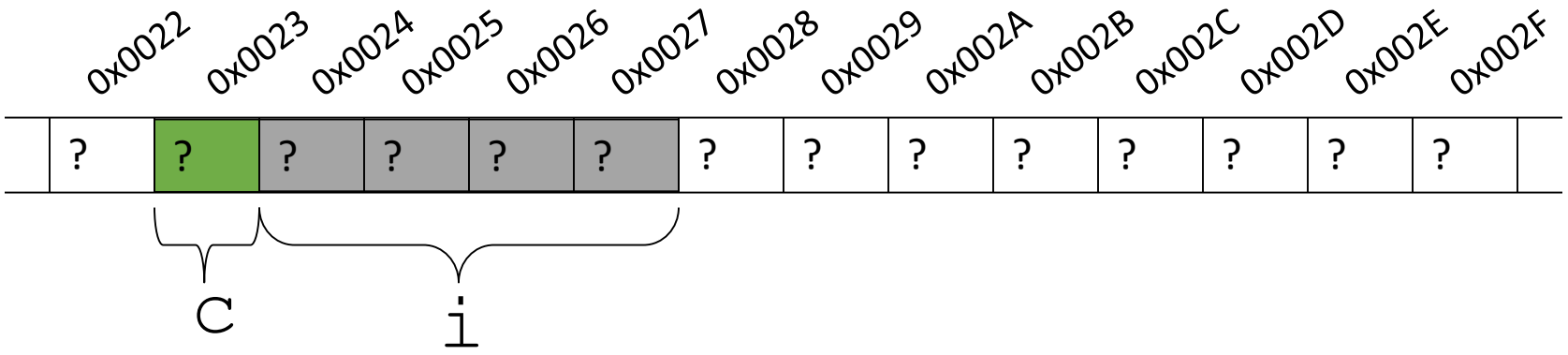


Memory elements have unique number (address)

Sequentially ordered

Memory element has stored value (content)

Variable Declaration



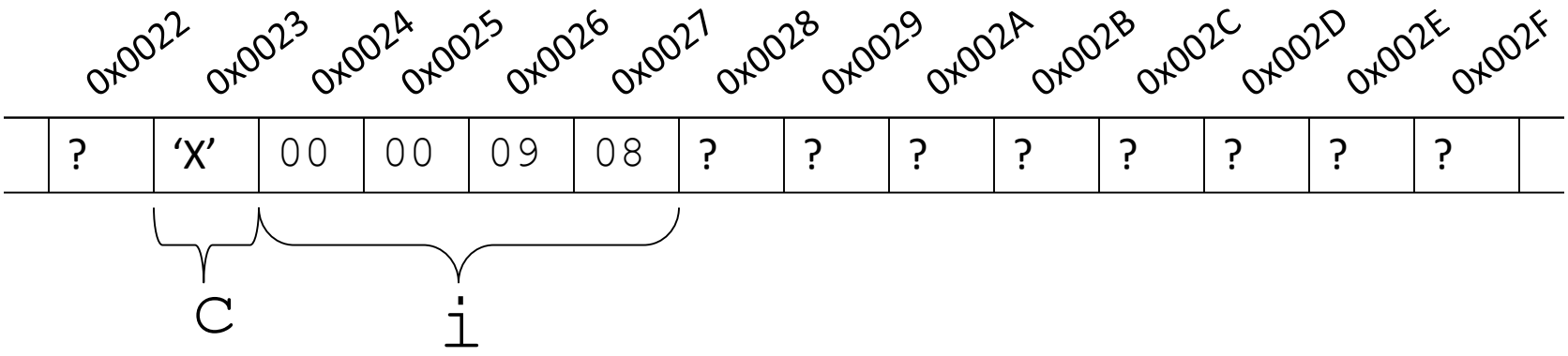

```
char c;
```

`c` is at address 23

```
int i;
```

`i` is at address 24 (start)

Variable Assignment



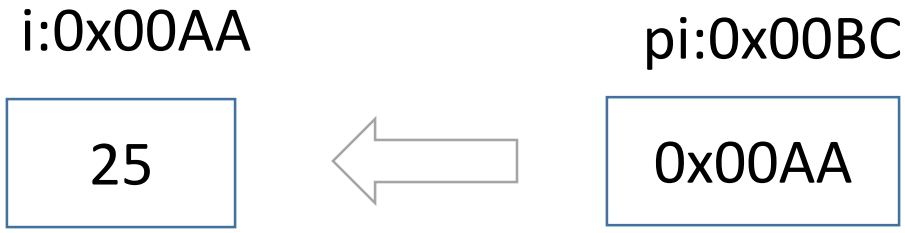

```
c = 'X';
i = 00 00 09 08;
```

Store `'X'` at address 23

Store `0x00000908` at address 24

Big-Endian Notation!
First byte is most significant,
last byte least significant.

Pointer Intro



- Pointer is a variable holding an address to a memory location.
  - 4 bytes in size for 32-bit machine
  - 8 bytes in size for 64-bit machine
- Changing a pointer value means changing the memory address the pointer “points” to.

Pointer Assignment & Dereferencing

Declaring pointers:

```
int i, j;      // declares two integer variables
int *pi, *pj; // declares pointers to integer variables

i = 10; j = 5;
```

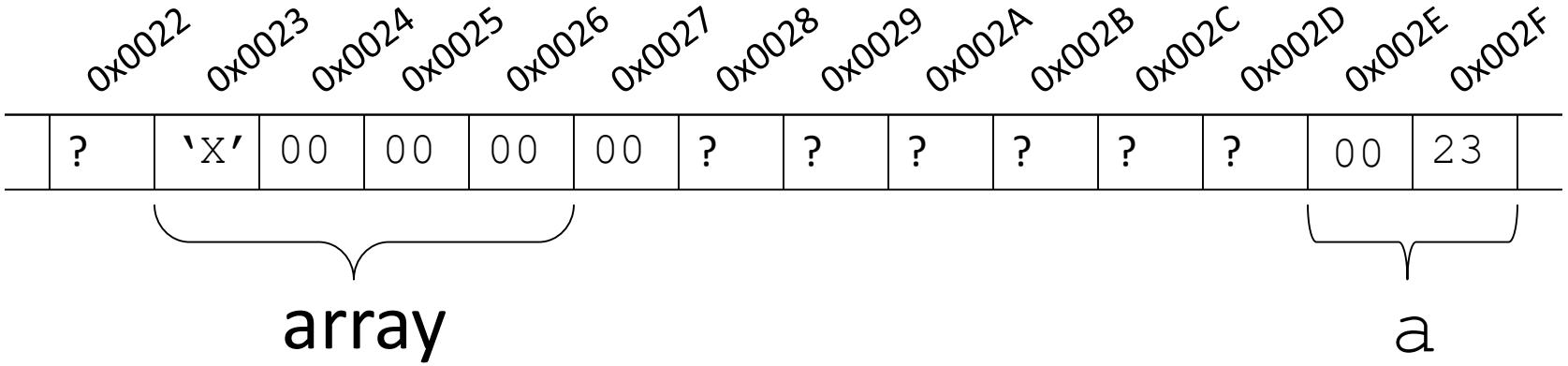
Assigning pointer a value:

```
pi = &i;
pj = &j;
```

Dereference pointer:

```
printf ("i = %d", *pi);
Printf ("j = %d", *pj);
```

Array Declaration




```
char a[4];
```

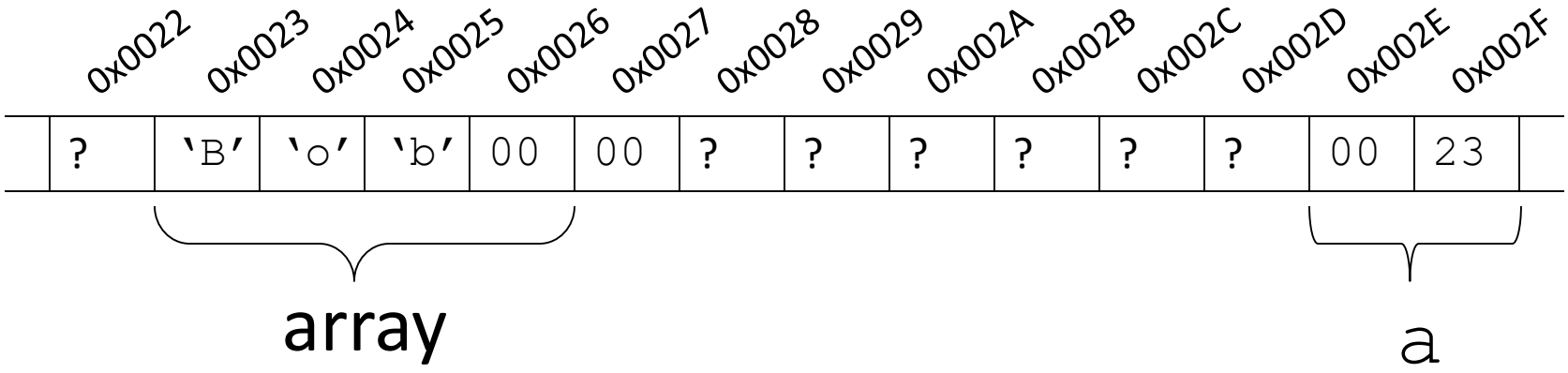
Allocate elements for 4 `chars`.

Allocate element for pointer to array.

Store start address of array in pointer a.

Array name ‘a’ is really pointer to 1st element!

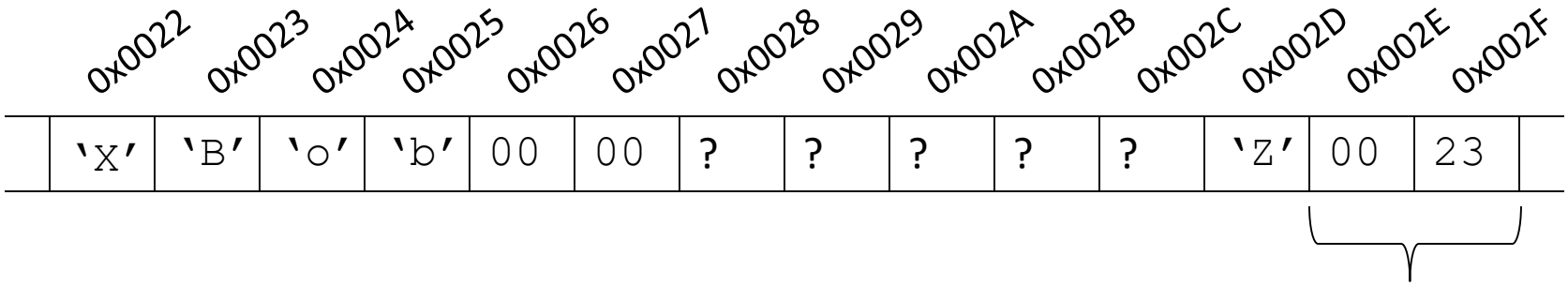
Array Assignment




```
a[0] = 'B';
a[1] = 'o';
a[2] = 'b';
```

Using array index to specify element

Array Assignment




```
a[-1] = 'X';
a[10] = 'Z';
```

Array size only for compile

No bounds checking – no bounds at runtime

These are legal statements!

Array Assignment (Code)

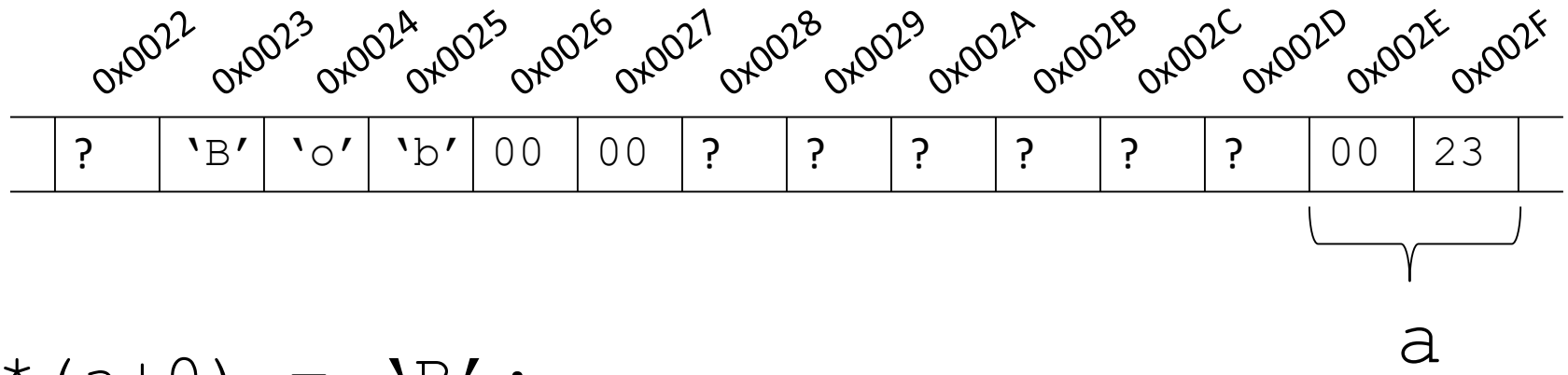
Declaring an array and assigns values:

```
// declares an array of chars and initialize it
char text[] = "Hello, World!";

// declares an array of numbers and initializes it
int numbers[] = {5, 4, 3, 2};

// declares an array of of points
Point allPoints[] = {p1, p2}; // p1, p2 are of type Point
```

Dereference pointer:

```
printf ("character = %c", *text);
printf ("number = %d", *number);
Printf ("number = %d", number[0]);
```

Array Arithmetic & Assignment

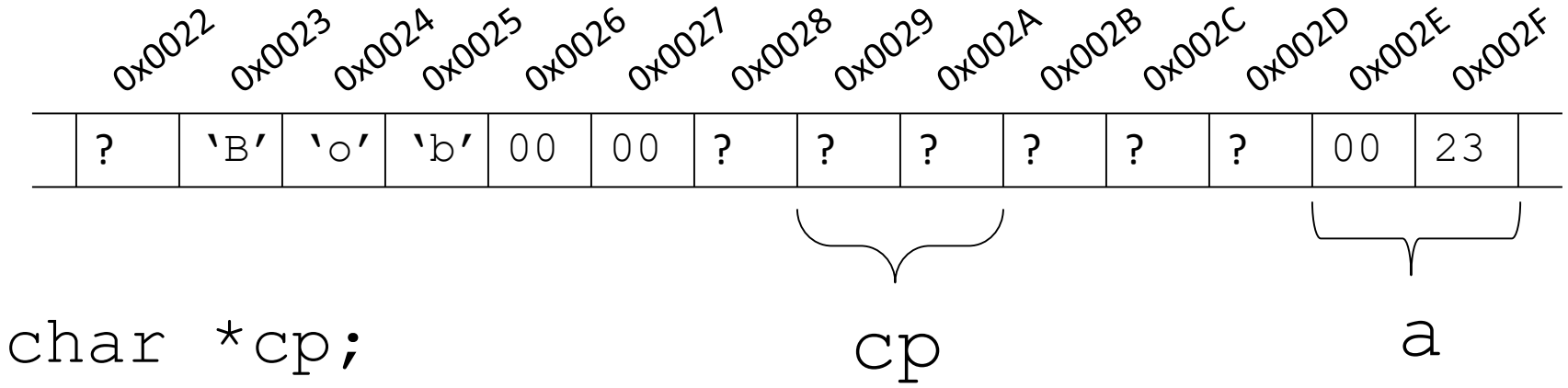



```
*(a+0) = 'B';
*(a+1) = 'o';
*(a+2) = 'b';
```

Using a pointer to specify element

* - “follow the pointer”

Memory Management



Allocates memory for the pointer (address)

`*` - “a pointer to”

No allocation for data, just a pointer!

Allocating & Freeing Memory

Static memory allocation:

```
int a[10];      // declares a static array for 10 ints
char name[64]; // declares a static array for 64 chars
```

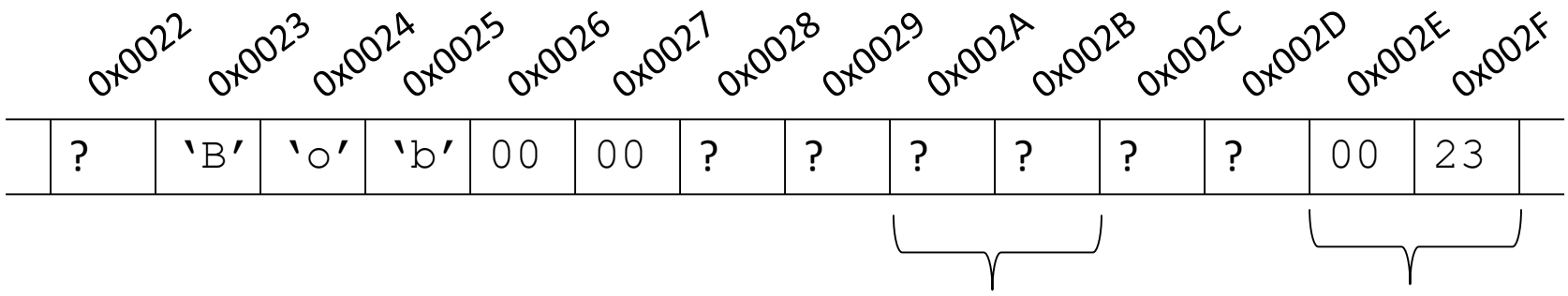
Dynamic memory allocation:

```
char *name = (char*) malloc (64*sizeof(char));
int *a = (int *) malloc (10*sizeof(int));
```

Freeing memory of dynamically allocated memory:

```
free (name);
free (a);
```

Assigning Array Pointers



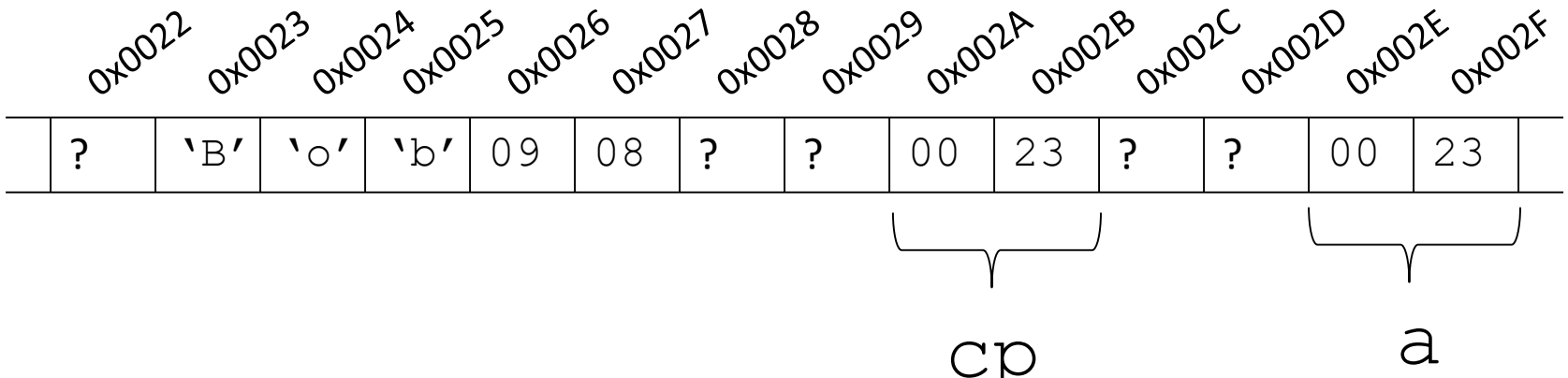

```
cp = a;
```

Copy an address into the new pointer

`&` - "the address of"

Could also use `cp = &(a[0]);`

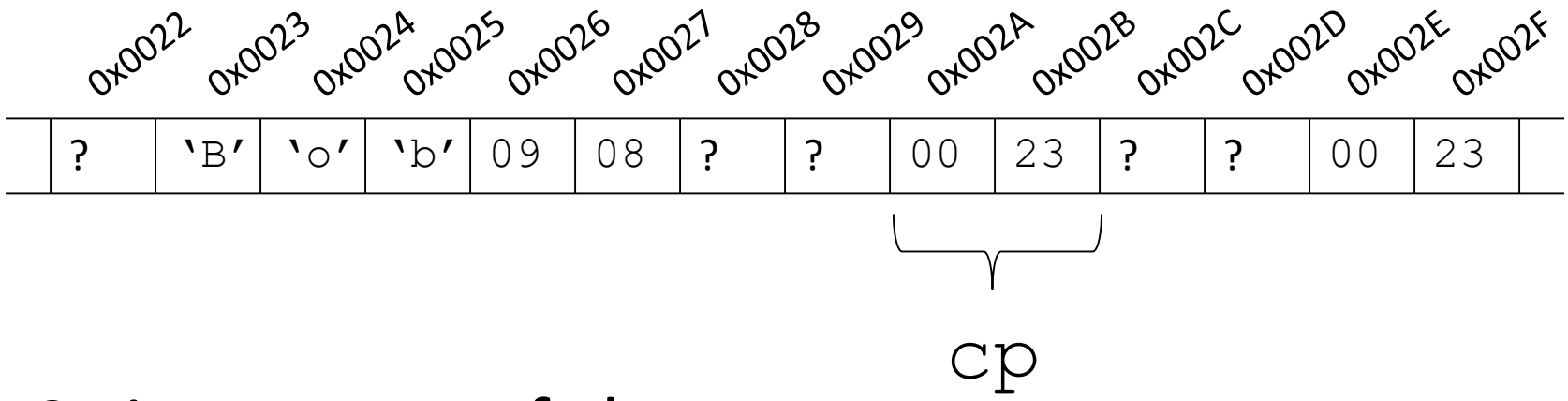
Using Pointers




```
printf("%c", *(cp+1);
printf("%c", cp[-1]);
printf("%c", *cp);
```

Can use array index or pointer math

Strings



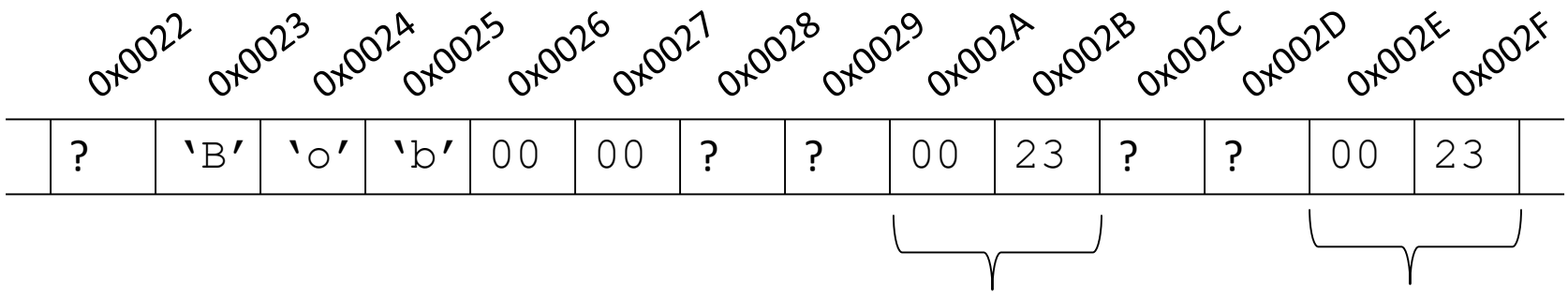
String ≈ array of char

Functions only need start address

Size of array not retained after compilation

How do functions find the end of the string?

String Functions




```
printf("%s", cp);
```

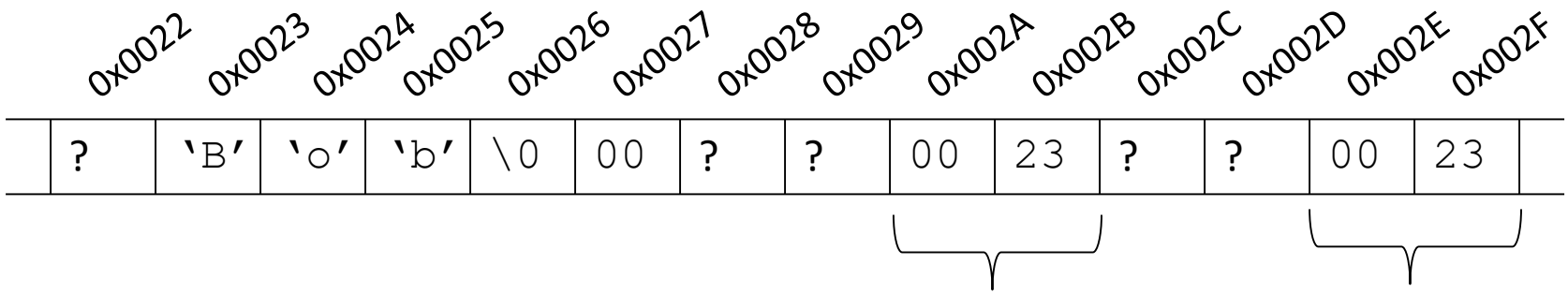
Function knows where 1st char is (`cp`)

Where is last char in string?

Size of array not known at runtime

Size may not equal string length

End-Of-String Marker




```
cp[3] = '\0';
```

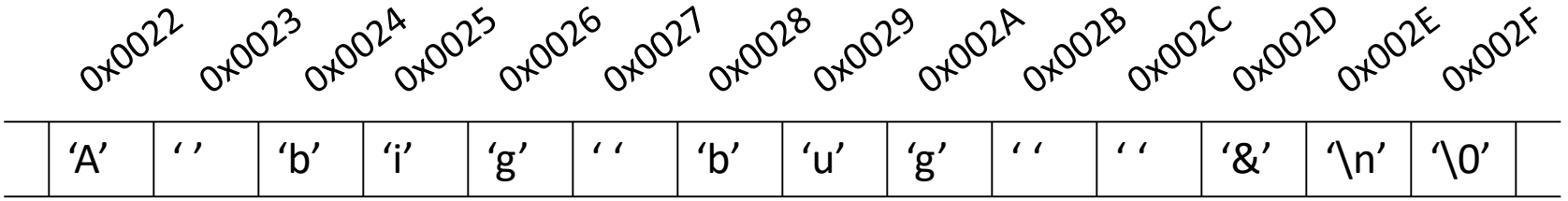
Special EOS character (`'\0'`)

Marks end of string location

`printf()` puts chars on screen until EOS

Forget the EOS and garbage will print

String Tokenizer, 1



Assume `char buff[256];`

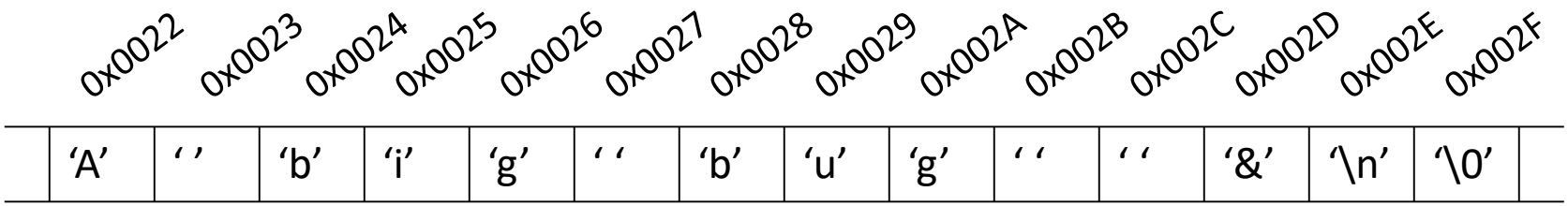
`buff` contains address 22

Read "`A big bug  &`" from *stdin* into buff

Assume `char *sp[5];`

- An array of 5 strings
- No memory allocated to save characters

String Tokenizer, 2



Tell `strtok()`

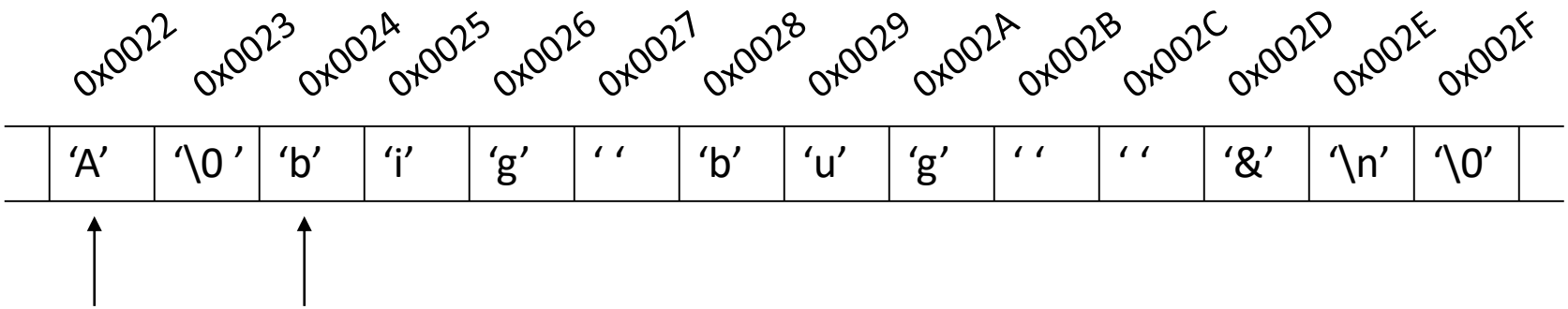
where to start parsing

what to look for (delimiters)

```
strtok(buff, " \t\n");
```

Space, tab and return are common

String Tokenizer, 3

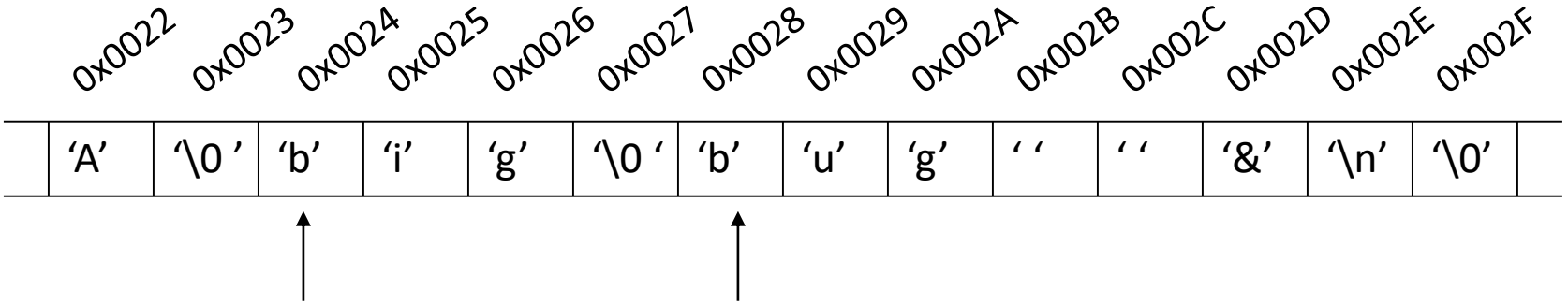



```
sp[0] = strtok(buff, " \t\n");
```

What `strtok()` does

- replaces 1st delimiter with EOS
- returns address of 1st non-delimiter
- remembers where it stopped

String Tokenizer, 4




```
sp[1] = strtok(NULL, " \t\n");
```

After 1st call, tell `strtok()` to continue

Give `NULL` as 1st parameter

Same functionality, but doesn't restart

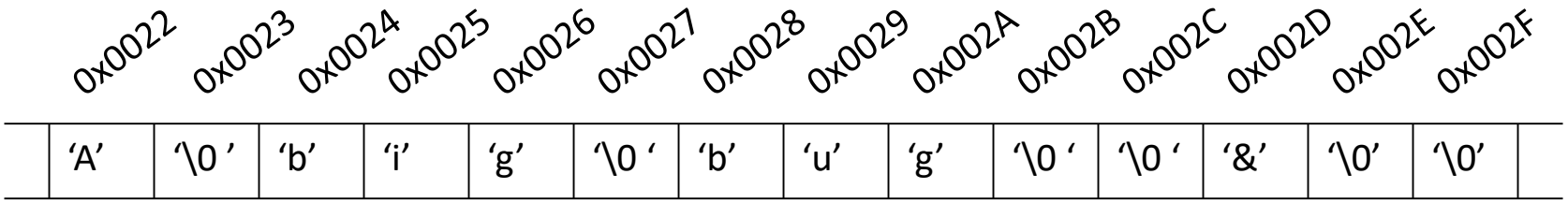
String Tokenizer, 5

	0x0022	0x0023	0x0024	0x0025	0x0026	0x0027	0x0028	0x0029	0x002A	0x002B	0x002C	0x002D	0x002E	0x002F	
	'A'	'\0'	'b'	'i'	'g'	'\0'	'b'	'u'	'g'	'\0'	'\0'	'&'	'\0'	'\0'	

```
sp[2] = strtok(NULL, " \t\n");
sp[3] = strtok(NULL, " \t\n");
sp[4] = strtok(NULL, " \t\n");
```

Returns NULL when no more tokens (EOS)

String Tokenizer, 6




```
sp[0]    0x0022
sp[1]    0x0024
sp[2]    0x0028
sp[3]    0x002D
sp[4]    NULL
```

Normally used in while loop

While not `NULL`…

MAN Page, Top

NAME

strtok strtok_r – extract tokens from strings

SYNOPSIS

```
#include <string.h>
char *strtok(char *str, const char *delim);
char *strtok_r(char *str, const char *delim, char **saveptr);
```

DESCRIPTION

The **strtok**() function parses a string into a sequence of tokens. On the first call to **strtok**() the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str should be NULL.

The delim argument specifies a set of characters that delimit the tokens in the parsed string. The caller may specify different strings in delim in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting character. If no more tokens are found, **strtok**() returns NULL.

MAN Page, Bottom

…

BUGS

Avoid using these functions. If you do use them, note that:

These functions modify their first argument.

These functions cannot be used on constant strings.

The identity of the delimiting character is lost.

The **strtok**() function uses a static buffer while parsing, so it's not thread safe. Use **strtok_r**() if this matters to you.

…

SEE ALSO

index(3), **memchr**(3), **rindex**(3), **strpbrk**(3), **strsep**(3), **strspn**(3), **strstr**(3)