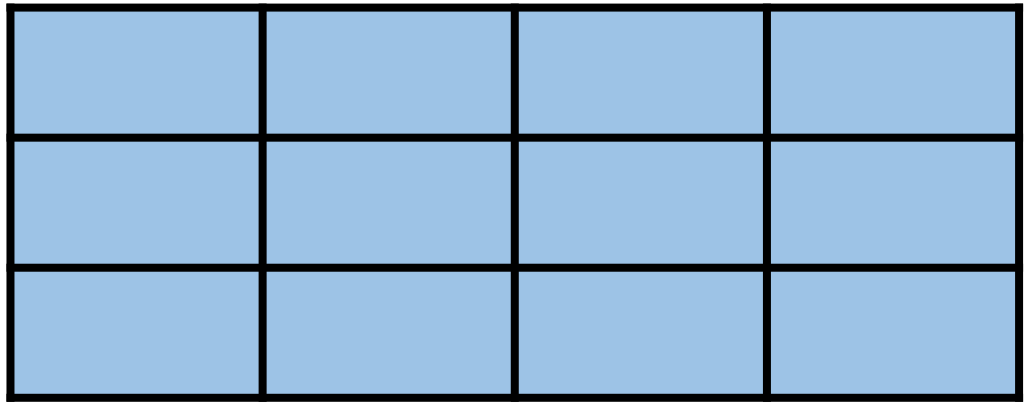
iClicker Attendance

Please click on A if you are here:

A. I am here today.

Pointers vs. Arrays



Array Values are "Contiguous"

- Right next to each other in memory
- int vec[6]

vec[0]	vec[1]	vec[2]	vec[3]	vec[4]	vec[5]

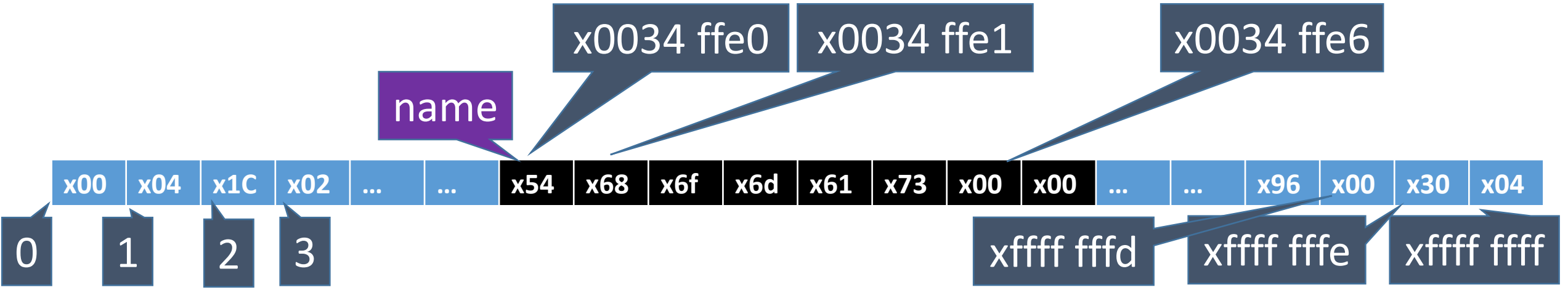
- int m [4][3];

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]	m[2][0]	m[2][1]	m[2][2]	m[3][0]	m[3][1]	m[3][2]

Example Array in Memory

```
char name[8]="Thomas";
printf("name is at %p\n",&name[0]);
```

name is at 0x34ffe0



Pointer to Array in Memory

```
char name[8]="Thomas";
char *np=&name[0];
printf("np is %p\n",np);
```

np is 0x34ffe0

iClicker Question

```
char name[8]="Thomas";
char *np=&name[0];
```

- What does np point to?

A. A single character
B. An array of characters
C. A string
D. All of the above
E. None of the above

What are we Pointing At?

```
char name[8]="Thomas";
char *np=&name[0];
printf("np -> %c\n",(*np));
```

np -> T

Question:
Does np point to a single character?

Pointers are just numbers

```
char name[8]="Thomas";
char *np=&name[0];
printf("np=%x np+1=%x np+2=%x",np,np+1,np+2);
```

np=x34ffeo np+1=x34ffe1 np+2=x34ffe2

What are we Pointing At?

```
char name[8]="Thomas";
char *np=&name[0];
printf("np ->%c-%c-%c...\n",*np,*(np+1),*(np+2));
```

np -> T-h-o...

Question:
Does np point to a single character?
Or an array of characters?

What is a "string"?

- A "string" is just a vector of ASCII characters
  - Followed by a "null terminator" – a byte with the value 0x00

char str[14]="This a string";

{'T', 'h', 'i', 's', ' ', 'a', ' ', 's', 't' ,'r' ,'i' ,'n' ,'g', x00}

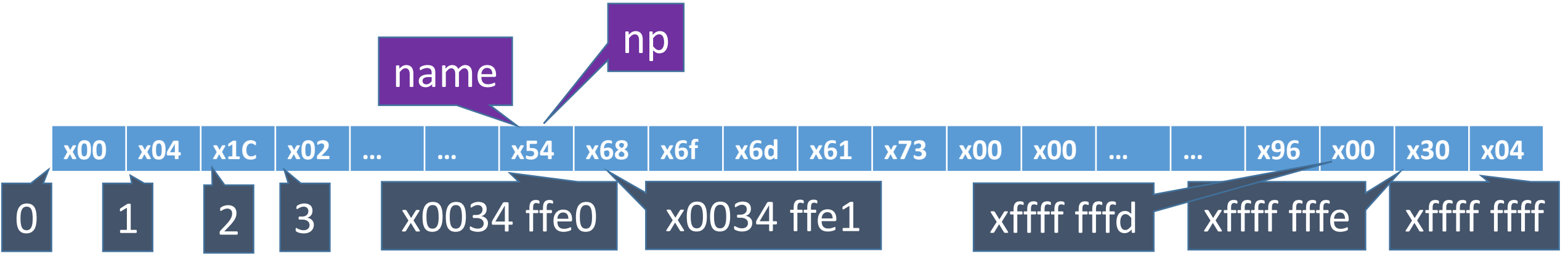
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
ASCII	T	h	i	s		a		s	t	r	i	n	g	
Hex	x54	x68	x69	x73	x20	x61	x20	x73	x74	x72	x69	x6e	x67	x00

What are we Pointing At?

```
char name[8]="Thomas";
char *np=&name[0];
printf("np ->%s\n",np);
```

np -> Thomas

Question:
Does np point to a single character?
Or an array of characters?
Or a string?



A Pointer points to one or more elements of a specific type

(Actually, zero or more, but who's counting)

Integer Vector in Memory

```
int v[4]={11,12,13,14};
```

Vector	v[0]	v[1]	v[2]	v[3]
Value	x00 00 00 0B	x00 00 00 0C	x00 00 00 0D	x00 00 00 0E
Address	x00 c0 00 14	x00 c0 00 18	x00 c0 00 1c	x00 c0 00 20

```
int *vp=&v[0]; // vp=x00c0 0014
printf("vp-> %d %d %d ... \n",*vp,*(vp+1),*(vp+2));
```

```
vp-> 11 12 13 ...
```

Pointer Arithmetic

In C, if we add "1" to a pointer, that means, point to the next element

- If `char * ptr` points to a character,
  then `ptr+1` points to the next character

- If `int * iptr` points to an integer,
  then `iptr+1` points to the next integer

- If `float *aptr[4]` points to an array of 4 floats,
  then `aptr+1` points to the next array of 4 floats.

Pointer Arithmetic

In order to make this work, C uses "funny" arithmetic on pointers

- If `char * cptr` points to a character,
  `printf("char@ %p, next char@ %p\n",cptr,cptr+1);`
  prints: char @ 0xffffcc0f, next char @ 0xffffcc10

- If `int * iptr` points to an integer,
  `printf("int @ %p next int @ %p\n",iptr,iptr+1);`
  int @ 0xffffcc08, next int @ 0xffffcc0c

- If `int **pptr` points to a pointer which points to an integer,
  `printf("pointer @ %p next pointer @ %p\n",aptr,aptr+1);`
  pointer @ 0xffffcc00, next pointer @ 0xffffcc08

Pointer Arithmetic

- At first, this seems … well just weird
- But soon it becomes clear… this is VERY powerful!

```
vec[3] == *(vec+3) // No matter what type elements are!
```

2D Arrays in Memory

```
int m[4][3]={{11,12,13},{21,22,23},{31,32,33},{41,42,43}};
```

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]	m[2][0]	m[2][1]	m[2][2]	m[3][0]	m[3][1]	m[3][2]
11	12	13	21	22	23	31	32	33	41	42	43
c0 0014	c0 0018	c0 001c	c0 0020	c0 0024	c0 0028	c0 002c	c0 0030	c0 0034	c0 0038	c0 003c	c0 0040

```
int *mp=&m[0][0]; // mp=x00c0 0014
printf("mp-> %d %d %d ... \n",*mp,*(mp+1),*(mp+2));
printf("row 3: %d %d %d\n",*(mp+6),*(mp+7),*(mp+8));
```

```
mp-> 11 12 13
row 3: 31 32 33
```

What is an un-sub-scripted Array?

```
char name[8]="Thomas";
printf("name[0] is at %p\n",&name[0]);
printf("name value is %x\n",name);
```

name[0] is at 0x23cb10

name value is 23cb10

array == &array[0]

What is a String?

```
char name[8]="Thomas";
printf("name[0] is at %p\n",&name[0]);
printf("name value is %x\n",name);
printf("name string is %s\n",name);
```

name[0] is at 0x23cb10

name value is 23cb10

name string is Thomas

A string is a pointer to zero or more characters

with a null terminator

What is a String?

```
char *name="Thomas";
printf("name[0] is at %p\n",&name[0]);
printf("name value is %x\n",name);
printf("name string is %s\n",name);
```

```
name[0] is at 0x23cb10
name value is 23cb10
name string is Thomas
```

In C, Pointers and Arrays are Virtually Interchangeable!

Array vs. Pointer Notation

Array	Pointer
&array[0]	array
array[i]	*(array+i)

strlen implementations

Using array notation

```
int strlen(char str[]) {
   int i=0;
   while(str[i]!=x00) i++;
   return i;
}
```

Using pointer notation

```
int strlen(char *str) {
   int i=0;
   while((*str)!=x00) { i++; str++; }
   return i;
}
```

Pointers and Memory

- When you declare an int or array of ints, the compiler reserves memory for that int or array of ints in the invocation record
- When you declare a pointer to an int or array of ints, the compiler reserves memory for the POINTER, but does NOT reserve memory for the actual data being pointed to!
- Typical C bug:

```
int char* name; // name is an uninitialized pointer
strcpy(name,"Tom Bartenstein");
```

Segmentation Violation

Initialized Pointers

- In some cases, it *looks* like pointers ARE reserving memory
  `char * name = "Tom Bartenstein";`

- In this case, the compiler is creating a LITERAL value in memory
  - Possibly "read only" memory
  - Possibly re-used with other instances of "Tom Bartenstein" in this code

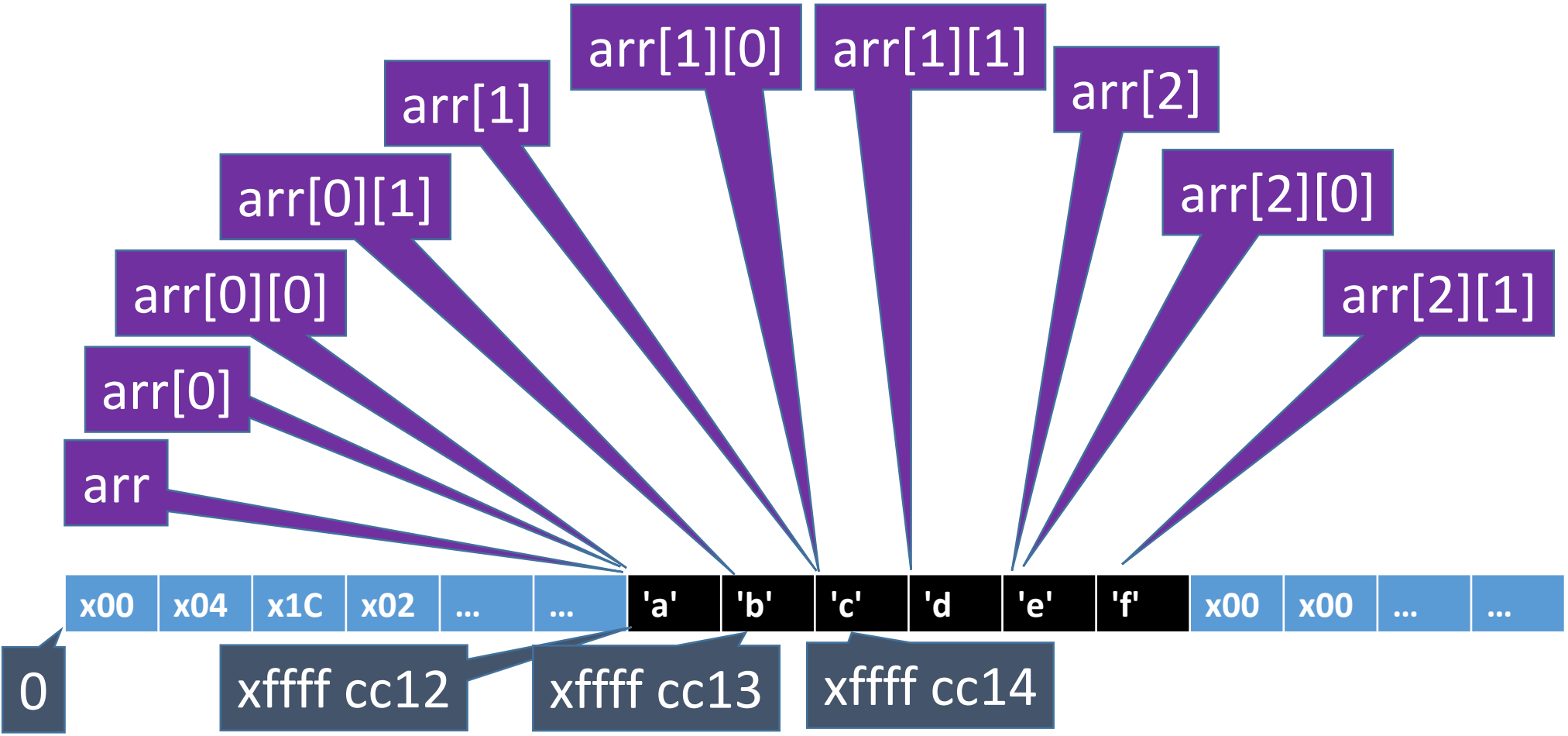
- The "name" variable is a pointer to the LITERAL value
  - May get segmentation violation if written to
  - May get bizarre future errors if compiler re-used this literal

Matrix vs. Array of Arrays

- A matrix is a two-dimensional list of elements in which all values are contiguous in memory in row-major order
- An array of arrays is a list of elements in which each element is a pointer to another list of elements

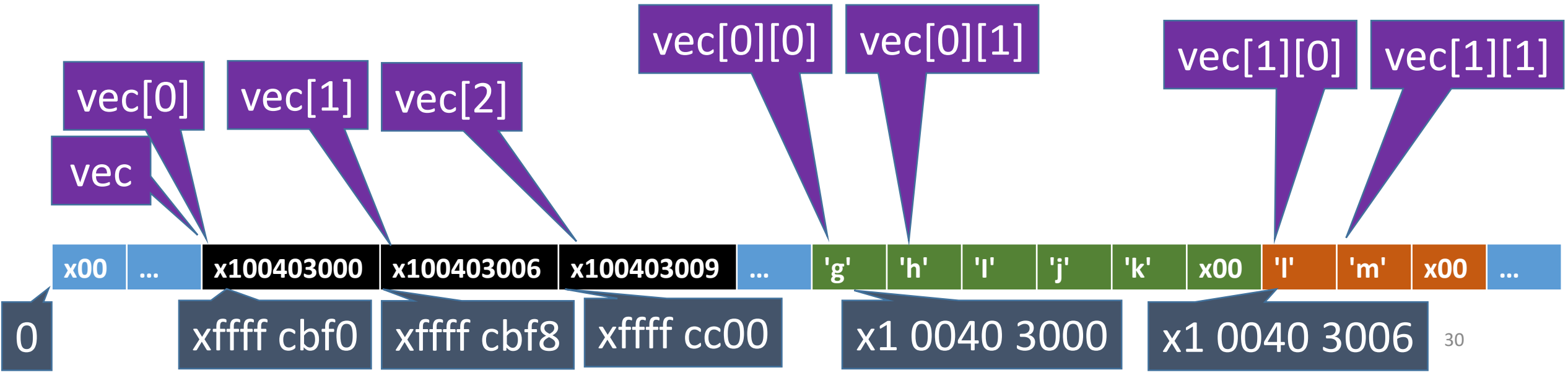
Matrix

char arr[3][2]={"ab","cd","ef"};



Array of Arrays

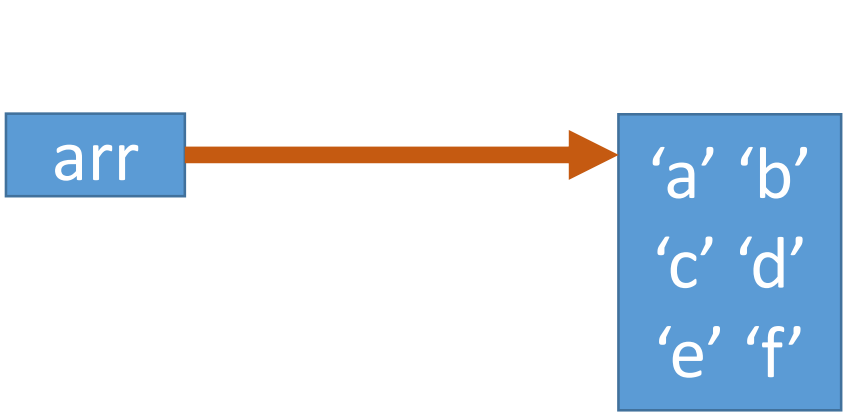
```
char *vec[3]={"ghijk","lm","nopqr"};
```

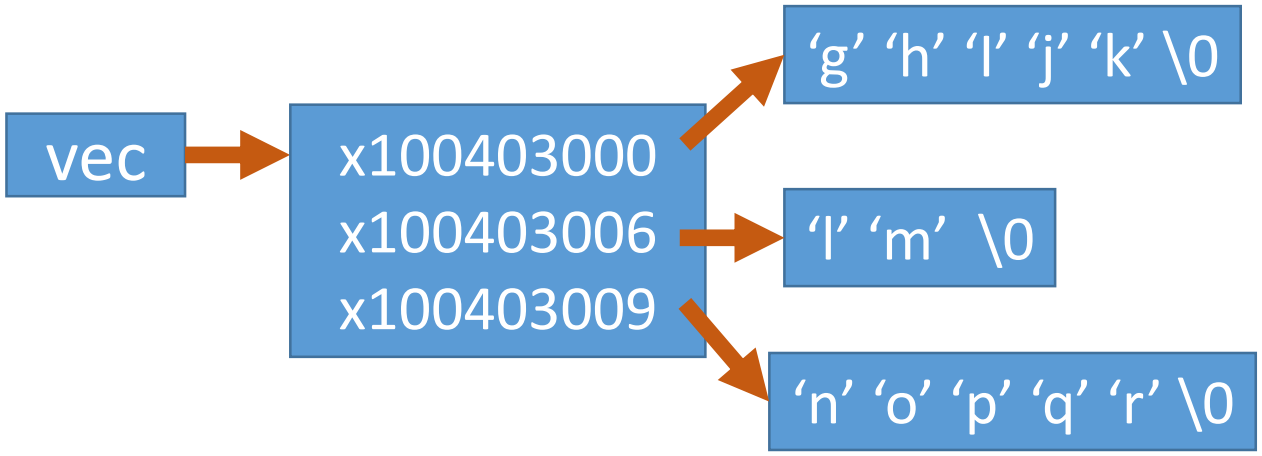


Or – another visualization

```
char arr[3][2]={"ab","cd","ef"};
```

```
char *vec[3]={"ghijk","lm","nopqr"};
```





Void Pointers

```
void * myptr; // myptr is a pointer to void
```

- myptr is a pointer, but I'm not going to tell you what it points at
- Before you use myptr, you must cast it as a pointer to something

```
printf("myptr points to %c\n",*(char *)myptr);
```

- void * used as a "universal pointer" – a pointer to any type of data
- Programmer must know what type of data it's pointing at to cast correctly

Resources

- Programming in C, Chapter 10
- Wikepedia Pointers : https://en.wikipedia.org/wiki/Pointer_(computer_programming)
- C Pointer Tutorial : http://www.tutorialspoint.com/cprogramming/c_pointers.htm