Arrays in C

Abstract Vector

Chap 3.8

- One dimensional array
- Ordered list of values, all of the same type
- Individual elements accessible by “index”
- Vector has a Size (Number of elements)

0	1	2	3	4	5
17.3	14.5	3.2	12.0	5.65	14.5

C Array Implementation

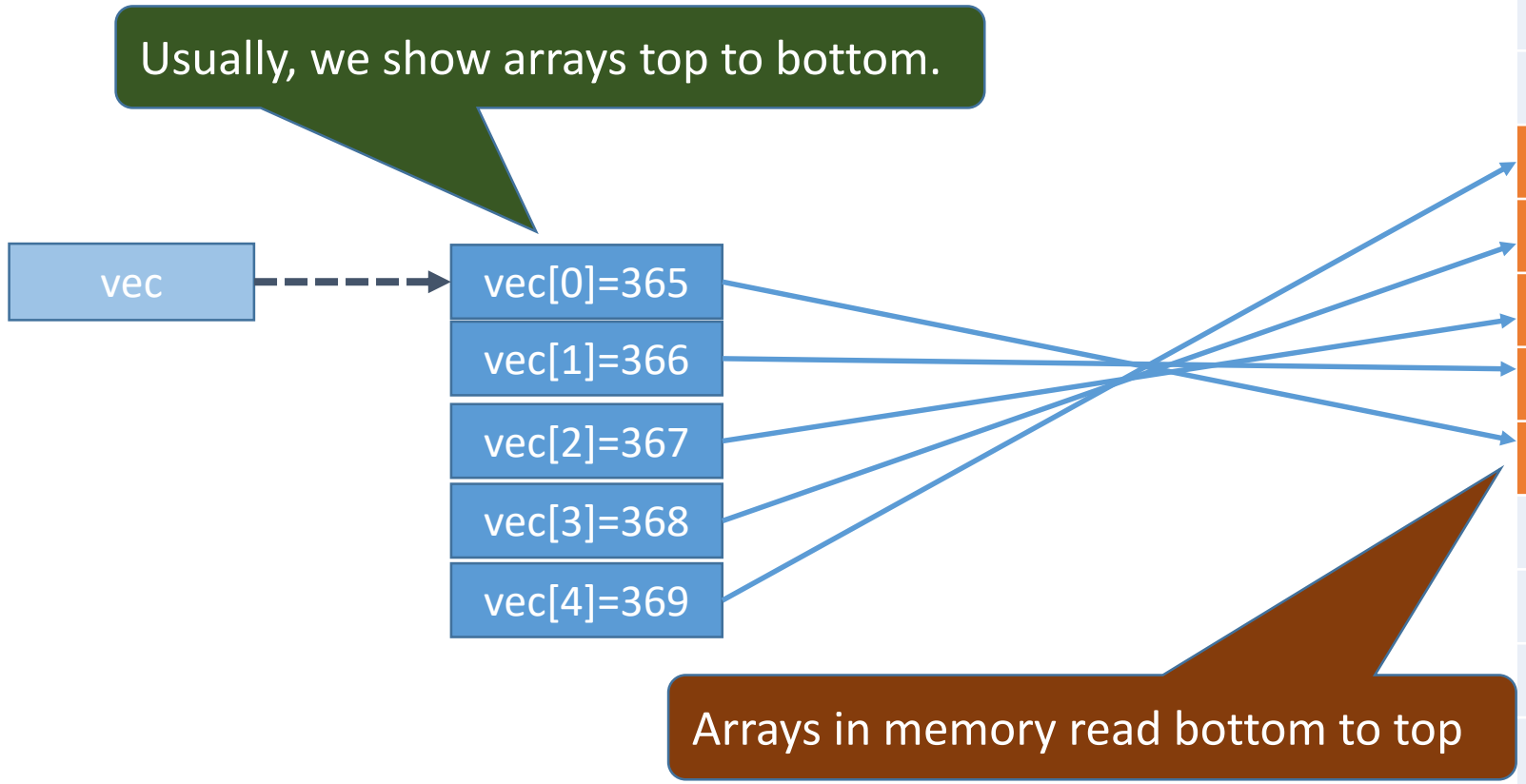
- List of contiguous values in memory
- Array Declaration:

- Type: Type of each element
- Name: Identifier for the entire array
- Count: Number of elements in the list

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	…	
vec[4]	0x0000 0C14	0x1111 1111
vec[3]	0x0000 0C10	0x1111 1111
vec[2]	0x0000 0C0C	0x1111 1111
vec[1]	0x0000 0C08	0x1111 1111
vec[0]	0x0000 0C04	0x1111 1111
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Inverted Arrays

```
int vec[5]={365,366,367,368,369};
```



Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	0xFFFF FFF0	0xDEAD BEEF
	…	
vec[4]	0x0000 0C14	0x0000 011C
vec[3]	0x0000 0C10	0x0000 011B
vec[2]	0x0000 0C0C	0x0000 011A
vec[1]	0x0000 0C08	0x0000 0119
vec[0]	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Initializing a Vector

```
int x=7; // Initializing a scalar variable
int gpc[12]={4,4,6,4,3,3,2,2,3,1,4,4}; // Initializing vector
```

0	1	2	3	4	5	6	7	8	9	10	11
4	4	6	4	3	3	2	2	3	1	4	4

- Or, let the compiler count the number of elements

```
int gpc[ ]= {4,4,6,4,3,3,2,2,3,1,4,4};
```

- ***IF ARRAY IS NOT INITIALIZED IT'S INITIAL VALUE IS UNKNOWN!***

C Idiom: Zeroing out an Array

- Initialization “rule”… if you specify an initializer that does not contain enough values, C will “pad” your initializer to the right with 0x00.

- Thus, to initialize an entire array to zero, just specify a single 0 value.

```
int countOranges[37] = { 0 }; // All 37 orange boxes are empty
```

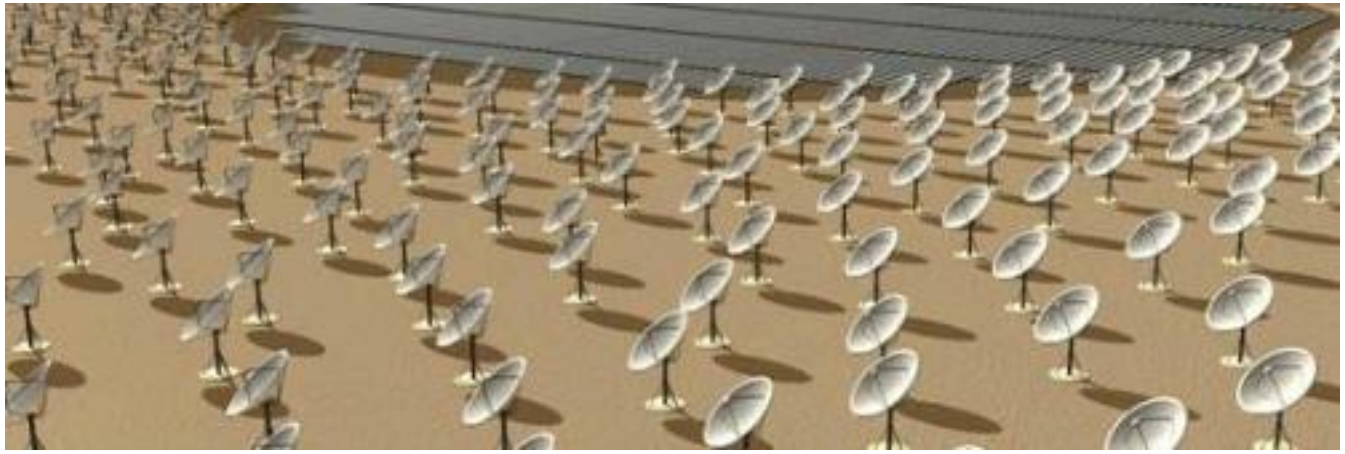
C Pitfall: Array Bounds Checking

```
int vec[5]; int i;
for(i=0;i<=5;i++) vec[i]=4;
```

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

- NO RUN-TIME ARRAY BOUNDS CHECKING IN C!!!!!!!!!!!
- Trust the programmer, and save the run-time!
- Programmer must be trustworthy!
- Writing past the end of an array can cause many problems
  - May write over other variables
  - May cause a segmentation violation

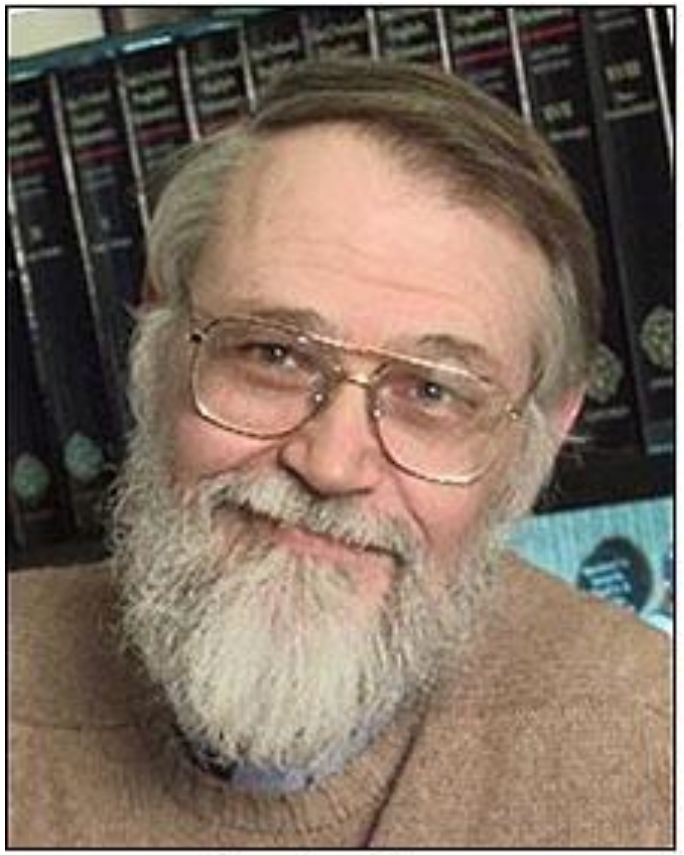
C Arrays and Pointers

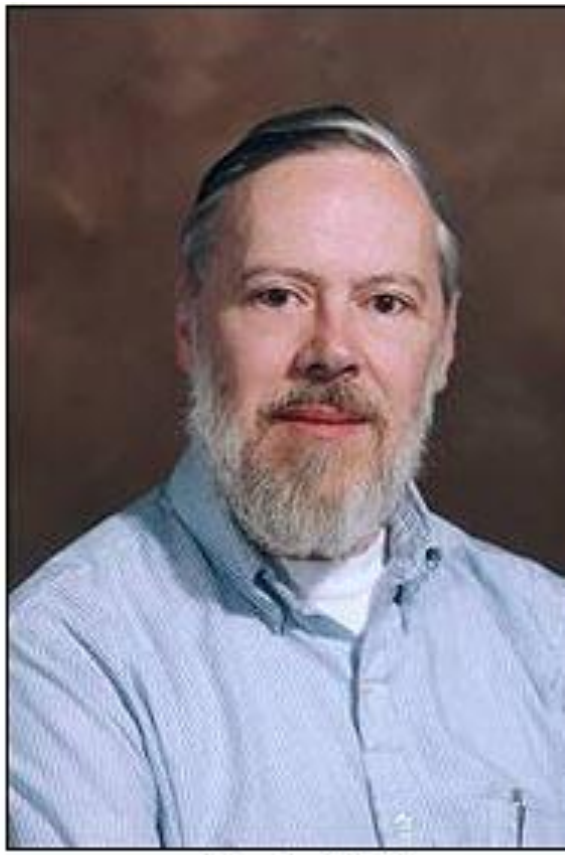


Arrays vs. Pointers…

"In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done by pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand".

K&R p. 97


Brian Kernighan


Dennis Ritchie

Array Name

- C convention: array name is the address of the first element!

$$vec == \&(vec[0])$$

- Therefore, the following holds:

$$\&(vec[i]) == (char\ *)vec + sizeof(vec[0]) * i$$

$$\&(vec[i]) = vec + i$$

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

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

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 (address+1)

- If `int * iptr` points to an integer,
  then `iptr+1` points to the next integer (address+4)

- If `float *aptr[4]` points to an array of 4 floats,
  then `aptr+1` points to the next array of 4 floats. (address+16)

Pointer Arithmetic Example

- A **"unit"** in pointer arithmetic is the size of the data type pointed to by the pointer

```
int *x; int vec[5];
for (x=vec; x<&vec[5]; x++) (*x)=3;
```

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	…	
x	0x0000 0D10	0x0000 0C18
	…	
vec[4]	0x0000 0C14	0x0000 0003
vec[3]	0x0000 0C10	0x0000 0003
vec[2]	0x0000 0C0C	0x0000 0003
vec[1]	0x0000 0C08	0x0000 0003
vec[0]	0x0000 0C04	0x0000 0003
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Arrays as Arguments

- C treats array arguments to pointers!
  - Call by reference means caller can see changes to arrays!!!
- For example, using string functions…

```
char name[100];
strcpy(name,"Tom Bartenstein");
int n=strlen(name);
scanf("%s ",name);
```

- If parameter dimensions are specified, compiler checks to make sure argument dimensions are correct

Can functions return Arrays?

- No!
- Functions can return arithmetic values, structures, or pointers, but NOT arrays!
  - However, it CAN return a pointer to an array..
- Problem: Where is the memory associated with a returned array?
  - If returned array passed in as an argument… that works
  - Local arrays – no longer valid after function call!
  - malloc'ed arrays – Fine as long as they eventually get freed
  - Static arrays – OK, but caller can't assume these are stable

Pointer / Array Ambiguity

- In C, we can treat pointers like arrays, and arrays like pointers

Using array notation

```
int suma(int nums[]) {
        int s=0; int i=0;
        while(nums[i]!=0) {
                s+=nums[i];
                i++
        }
        return s;
}
```

Using pointer notation

```
int sump(int *nums) {
        int s=0;
        while((*nums)!=0) {
                s+=(*nums);
                nums++;
        }
        return s;
}
```

Pointer/Array Ambiguity

- We may invoke functions which expect arrays by passing in pointers
- We may invoke functions which expect pointers by passing in arrays

```
int suma(int nums[]);
int sump(int *nums);
int numa[4]={1,2,3,0};
printf("sum=%d\n",suma(&numa[0]));
printf("sum=%d\n",sump(numa);
```

Array Dimensions

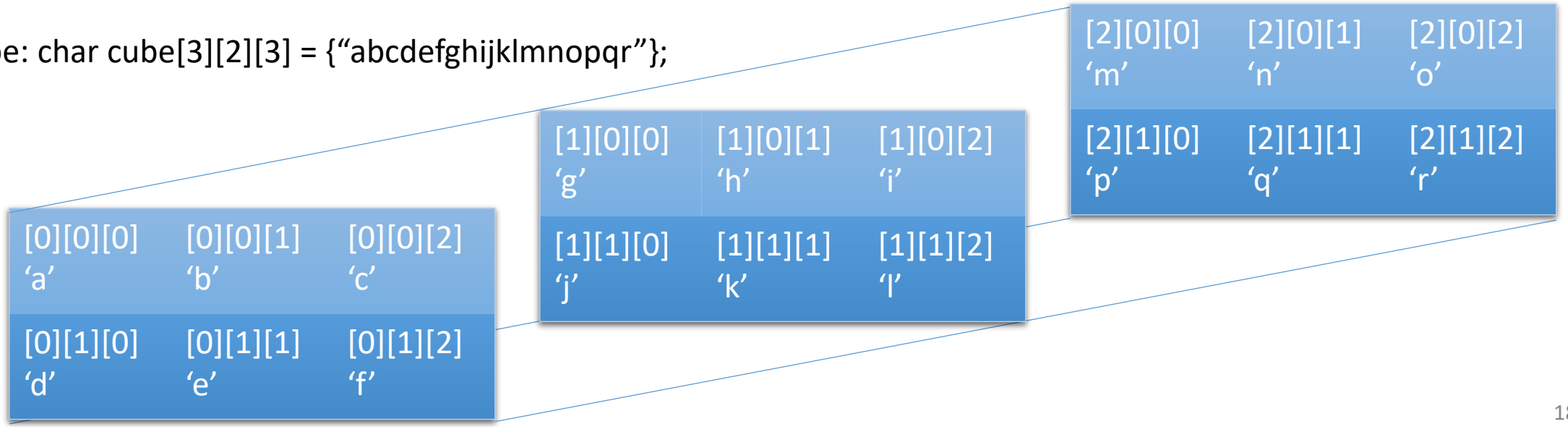
Vector: int vec[4]={10,20,30,40};

vec[0]	vec[1]	vec[2]	vec[3]
10	20	30	40

Matrix: int matrix[2][3]={{10,11,12},{20,21,22}}

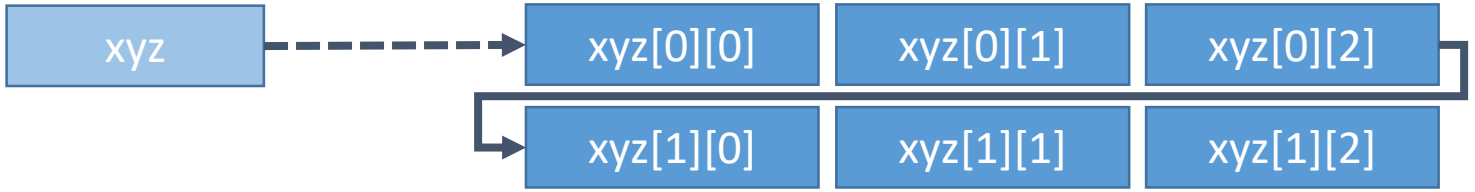
matrix[0][0] 10	matrix[0][1] 11	matrix[0][2] 12
matrix[1][0] 20	matrix[1][1] 21	matrix[1][2] 22

Cube: char cube[3][2][3] = {“abcdefghijklmnopqr”};



Multi-Dimensional Arrays

```
int xyz[2][3]; // 2 rows/3 cols
```



- Row Major order
- like odometer, right digit changes fastest

Label	Address	Value
	0xFFFF FFFC	0xDEAD BEEF
	0xFFFF FFF8	0xDEAD BEEF
	0xFFFF FFF4	0xDEAD BEEF
	…	
xyz[1][2]	0x0000 0C18	0x0000 011D
xyz[1][1]	0x0000 0C14	0x0000 011C
xyz[1][0]	0x0000 0C10	0x0000 011B
xyz[0][2]	0x0000 0C0C	0x0000 011A
xyz[0][1]	0x0000 0C08	0x0000 0119
xyz[0][0]	0x0000 0C04	0x0000 0118
	….	
	0x0000 000C	0x0000 000C
	0x0000 0008	0x0000 0009
	0x0000 0004	0x0000 0006
	0x0000 0000	0x0000 0003

Matrix Pointer Arithmetic

```
int mat[3][4]; int r; int c;
...
```

nCols

&(mat[r][c]) == mat + (r*4)+c

Or

elementSize=sizeof(mat[0][0]); rowSize=4*elementSize;

&(mat[r][c]) == (char *)mat + (r*rowSize) + c * elementSize

Base

row

table width

Offset

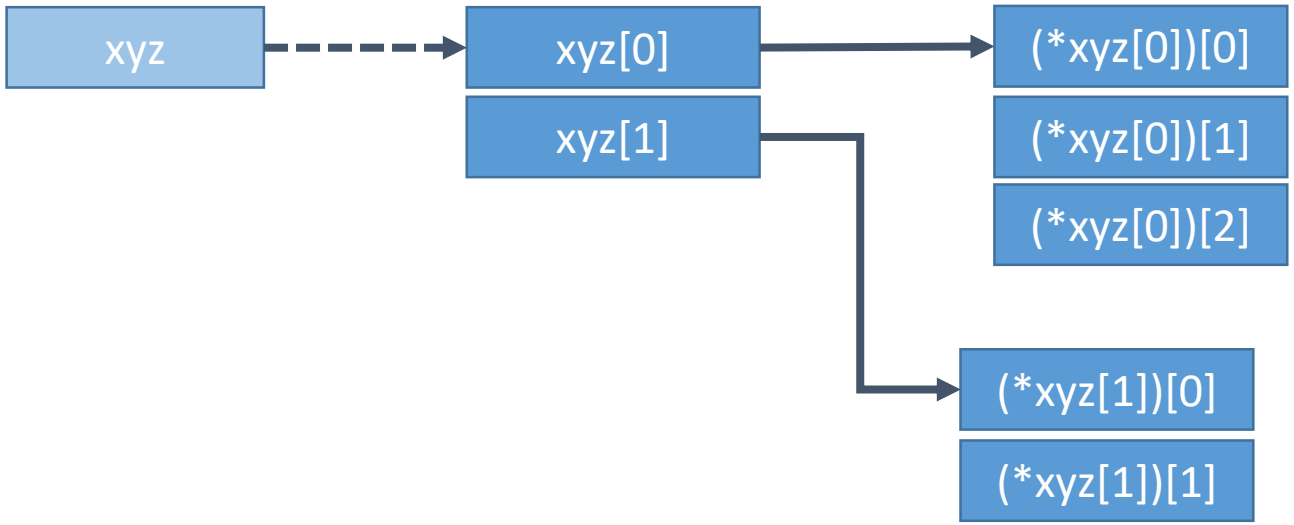
Higher Dimension Addresses

- With one dimension, “array” == “&array[0]”
- With two dimensions, “matrix” == “&matrix[0][0]”
  - Notice, &array[0] (or &matrix[0][0]) are not physically in memory!
- Also “matrix[i]” == “&matrix[i][0]”
  - matrix+i*sizeof(matrix[0])/sizeof(matrix[0][0])
  - Allows us to THINK of rows as sub-arrays
  - Again, there is no “matrix[i]” physically in memory!

- In 3d: “cube[i]” == “&cube[i][0][0]” and “cube[i][j]”== “&cube[i][j][0]”

Arrays of Pointers

```
int (*xyz)[2]; //Array of two pointers
xyz[0]=(int *)malloc(3*sizeof(int));
xyz[1]=(int *)malloc(2*sizeof(int));
xyz[1][0]=3;
…
```



Label	Address	Value
	…	
xyz[1]	0x0000 CA0C	0x0000 C708
xyz[0]	0x0000 CA08	0x0000 C000
	…	
xyz[1][1]	0x0000 C70C	0x0000 0004
xyz[1][0]	0x0000 C708	0x0000 0003
	…	
xyz[0][2]	0x0000 C008	0x0000 0002
xyz[0][1]	0x0000 C004	0x0000 0001
xyz[0][0]	0x0000 C000	0x0000 0000
	…	

Difference between Arrays and Pointers

Array (unsubscripted)

- Fixed, pre-defined dimension(s)
- Space reserved by compiler
  - In row major order
- All 2D rows equal length
- Not a variable… can't be re-assigned

Pointers to List

- No pre-defined length
- No space reserved
- 2D rows may be varying length
- Variable… can be re-assigned

Arrays with Undetermined Lengths

- In C, `int nums[];` is exactly the same as `int *nums;`
  - In both cases, `nums` is a pointer to an undetermined number of ints
  - In either case, we can use the notation `nums[3]` or `*(nums+3)` to retrieve the third element (if there is one)
- We must have some external way of knowing how many there are

- Note: For multi-dimensional arrays, only works for LEFTMOST index
  - e.g. `grades[][13]` is a pointer to a matrix that has 13 columns, but an unknown number of rows
  - `grades[20][]` is invalid! Why?

Dealing with Undefined lengths

Implicit	Explicit	Guard

int perimTri(int sides[]) {
  return side[0] + side[1] +
  side[2];
}
	int perimn(int n, int sides[]) {
	  int j; int sum=0;
	  for(j=0;j<n;j++)
	  sum+=sides[j];
	  return sum;
	}
		int perimg(int sides[]) {
		  int j; int sum=0;
		  for(j=0;sides[j]>0;j++)
		  sum+=sides[j];
		  return sum;
		}

• Agreement: sides is an array with three elements • Probably better to use "sides[3]" in argument definition	• Agreement: Caller specifies how many elements are in sides with an extra argument	• Agreement: Caller puts a guard value after last significant element of sides • Guard value is an "invalid" value

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 shared with other instances of "Tom Bartenstein" literal in this code

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

Strings in C

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

Pointers and Memory

- When you declare an int or array of ints, the compiler reserves memory for that int or array of ints
- 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:

```
char* name; // similar to char name[100] but…
strcpy(name,"Tom Bartenstein");
```

Segmentation Violation

Empty String

```
char str[14]="This a string";
str[0]=x00;
printf("Variable str contains <%s> and no more\n",str);
```

Variable str contains <> and no more

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

Standard library String Functions

- #include <string.h>
- strlen(string) – counts the number of characters in the string (up to null terminator)
- strcpy(from,to) – copies all characters in from string to to string
  - Assumes “to” is large enough to hold from string
  - If not sure, use strncpy(from,to,n)
- strcat(start,tail) – adds tail in place of null terminator to start
  - Assumes “start” is large enough to hold combined result
  - If not sure, use strncat(start,tail,n)
- strcmp(a,b) – compares a to b, returns -1 if a<b, 0 if a==b, 1 if a>b
  - ASCII order
  - Use strncmp(a,b,n) to compare prefixes

Pointer / Array Ambiguity

- In C, we can treat pointers like arrays, and arrays like pointers

Using array notation

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

Using pointer notation

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

Resources

- The C Programming Language, (K&R) Section 1.6 (Arrays), 1.9 (strings)
- Wikipedia C String Handling
  https://en.wikipedia.org/wiki/C_string_handling
- C String Tutorial :
  http://www.tutorialspoint.com/cprogramming/c_strings.htm