Arrays

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Declaring and using arrays
- Array initialization
- Operations on arrays
- Fixed- and variable-length arrays
- Arrays and sizeof()
- Multi-dimensional arrays
- Arrays and functions
- Strings as char arrays

Making Arrays

- When you declare an array, you get to specify
  - The type of its elements
  - How many elements it will have
  - How they will be initialized (if at all)
- Example:

```
int a[ 10 ];
```

- Notice, part of the type is given before the variable name, part after.
- In many contexts, the number of elements must be an integer constant expression

Meet the Array

- Array notation is similar to Java
- You can access element i in array a: `a[ i ]`
  - For an n-element array, indices range from `0` to `n-1`
- You can also say it backward: `i[ a ]`
  - Why can you do this?
  - It's a consequence of what the a[i] notation really means in C
  - We'll learn about this later

C vs Java

- In C, declaring an array variable allocates the array.

```
int a[ 10 ];
```

- In C, arrays can be allocated statically, on the stack or on the heap (later).

- In Java, you need to declare and allocate an array.

```
int[] a = new int [ 10 ];
```

- In Java, all arrays are always allocated on the heap.

Array Initialization

- Initialization rules are like other variables
  - If you don't initialize a **stack-allocated array**, it will contain whatever was previously in that memory.

```
int a[ 20 ];
for ( int i = 0; i < 20; i++ )
  printf( "%d ", a[ i ] );
```

outputs

0 1 1423550160 32767 182865920 …

Yay, garbage!

  - If you don't initialize a **statically-allocated array**, it starts out full of zeros.
- You can fill in the elements one at a time.

```
int a[ 20 ];
a[ 0 ] = 7;
a[ 1 ] = 14;
…;
```

Array Initialization

- Or, you can ask the compiler to initialize it for you:

```
int a[ 5 ] = { 7, 14, 21, 28, 35 };
```

- If you initialize just some of the array, the rest will be initialized to zero.

```
int a[ 5 ] = { 7, 14, 21 };
```

Imagine … , 0, 0

  - This is called *partial initialization*.

Array Initialization

- You can let the compiler determine the size
  - by measuring the length of your initialization list.

```
int a[] = { 7, 14, 21, 28, 35 };
```

OK. You must want a 5-element array.

- C99 offers an alternative initialization syntax

```
int a[] = { [7] = 10, [2] = 40, [5] = 35 };
```

0	0	40	0	0	35	0	10

Array Initialization

- You can even mix the two notations
  - Where you give an index, C will put the given value there.
  - Where there's no index, C will fill the subsequent index.
  - If you don't give an array size, you'll get an array that's just large enough to hold the last value.

```
int a[] = { 1, [7] = 10, 90, [2] = 40, 2,
            [5] = 35 };
```

1	0	40	2	0	35	0	10	90

Array Bounds Checking

- C does no bounds checking for array access
  - Like many things in C, this decision favors performance over security
  - And, in many situations, the compiler may not even know the size of the array anyway.

```
int a[ 6 ] = { };
```

- What do you get if you index off the end?
  - Let's find out.

Looking at Memory

- Example program howdyNeighbor.c

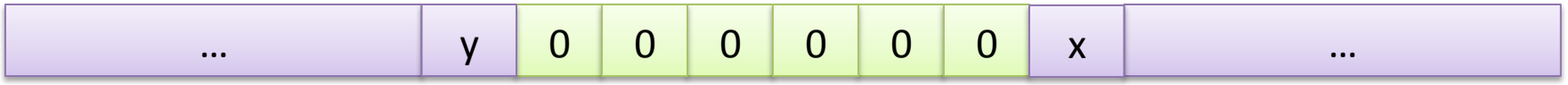
```
int x = 25;
int a[ 3 ] = { 10, 20, 30 };
int y = 50;

printf( "Array a: %d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );

printf( "A little before a: %d %d\n", a[ -2 ], a[ -1 ] );

printf( "A little after a: %d %d\n", a[ 3 ], a[ 4 ] );
```

- Output:

```
Array a: 10 20 30
A little before a: 4195712 50
A little after a: 25 1876150896
```

Looking at Memory

- From the example, I think this is what we're seeing:



- In this example, we're looking around in other parts of the *stack frame*.
  - Where the function stores its local variables and other values it needs.
- So every array is like a little window into our program's memory

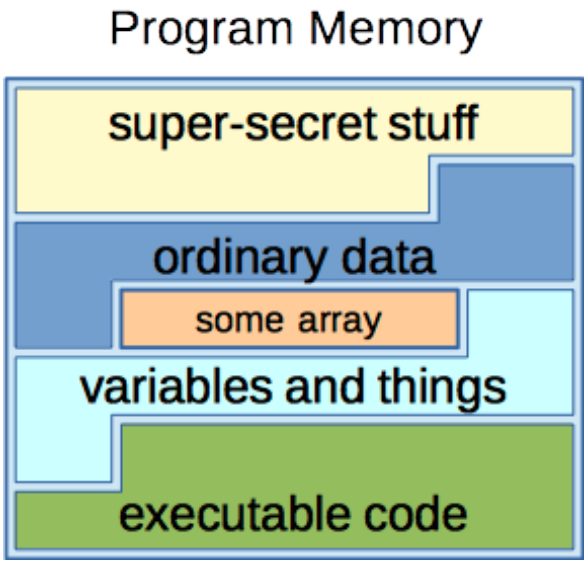
Array Bounds Checking

- Accessing memory outside an array's bounds can do lots of neat things.
  - Accidentally overwrite some of your program's variables
  - Segmentation fault, bus error.
- These errors can be difficult to debug.
  - Maybe you overwrite an important variable or something your program needs to keep running.
  - If you're lucky, this may crash your program right away.
  - If you're unlucky, your program may crash … later.

With Great Power …

- In Java, a bad index has defined behavior … it will throw an exception.
- In C, behavior is **undefined** … it could do anything … probably nothing good … for us.

- Accidentally writing past array bounds could be an exploitable vulnerability in our programs.
- An attacker could use this to change the data or behavior of a program.



Buffer Overflow

- A *buffer overflow* happens when a program writes into memory past the end of an array.
  - Typically while reading input from a user or another program.
- It's an easy mistake to make:

```
printf( "Please enter some numbers\n" );

int list[ 20 ];
int i;
while ( scanf( "%d", &list[ i ] ) == 1 )
    i++;
```

Operations on Arrays

- Normally, we work with an array one element at a time.
- C syntax includes very little for working with the whole array at once.
  - You can ask the size of an array:

```
printf( "%zd\n", sizeof( list ) );
```

  - But you can't copy from one array to another:

```
list1 = list2;
```

  - Or concatenate two arrays, etc.

Operations on Arrays

- Instead we typically need **loops** to process everything in an array:

```
int arrayCopy[ SIZE ];

for ( int j = 0; j < SIZE; j++ ) {
    arrayCopy[ j ] = arraySource[ j ];
}
```

Choosing Array Size

- The size of a statically allocated array must be a constant expression.

```
int sequence[ 100 ];

int main()
{
   ...
```

- This is true in some other contexts also.
- … and used to be true for stack-allocated arrays.

Variable-Length Arrays

- C99 permits the size of stack-allocated arrays to be determined during execution.

```
int main( void )
{
  int n;

  printf( "How many values would you like: " );
  scanf( "%d", &n );

  int list[ n ];

  ...
}
```

This is called a *variable-length array*.

What if this is zero?

or negative?

Variable-Length Arrays

- Variable-length arrays aren't resizable
  - Once you choose the size, you're stuck with it for the array's lifetime.

```
int n;
printf( "How many values? " );
scanf( "%d", &n );
int list[ n ];

… do some stuff…

printf( "Now how many? " );
scanf( "%d", &n );
int list[ n ];
```

This won't work; looks like you're trying to declare a new array.

Variable-Length Arrays

- But, you can choose a different size every time the variable goes into scope:

```
for ( int i = 0; i < 10; i++ ) {
    int n;
    printf( ”How many values? ” );
    scanf( "%d", &n );
    int list[ n ];

    ...;
}
```

This will work.

Determining Array Size

- You remember the `sizeof` operator … right?
- It reports the size of its argument, including array arguments.
  - An array's size is the number of elements times the size of each element.

```
#include <stdlib.h>

int a;
float b[100];
double c[ 16 ];

size_t aSize = sizeof( a );
size_t bSize = sizeof( b );
size_t cSize = sizeof( c );
```

Three variables

How much memory does each one take?

Computing Array Size

- The sizeof operator can let you figure out array length (number of elements).

```
float b[ 100 ];
…
int len;
len = sizeof( b ) / sizeof( b[0] );
```

It may not look like it, but this is really a constant expression.

Why not just say 100 instead?

This is a great trick … but it **doesn't work everywhere**.

- Here, sizeof() can be evaluated at compile time.

Multi-Dimensional Arrays

- To make a multi-dimensional array, you stack up square brackets after the name.
- Declared like this, you have to give a size for all dimensions.

```
int table[ 3 ][ 4 ];
```

Outer dimension
(normally we'd think of this as rows)

Inner dimension
(normally we'd think of this as columns)

- Accessing an element, just like you'd expect,
  - an index for row, then column, inside square brackets.

```
x = table[ 2 ][ 3 ];
table[ 1 ][ 0 ] = 99;
```

Multi-Dimensional Array Initialization

- With initialization, elements are initialized
  - row-by row, each row in curly brackets
  - and, left-to-right within each row.

0	1	4	9
16	25	36	49
64	81	100	121

```
int table[ 3 ][ 4 ] = {
  {0, 1, 4, 9},
  {16, 25, 36, 49},
  {64, 81, 100, 121 }
};

for( int i = 0; i < 3; i++ )
    for( int j = 0; j < 4; j++ )
        printf( "%d ", table[i][j] );
```

I'm the first row.

I'm the first element on the first row.

Prints 0 1 4 9 16 …

Partial Initialization

- You can do partial initialization in a 2D array

```
int table[3][4] = {
  { 0, 1 },
  { 16, 25, 36, 49 },
  { 64 }
};
```

You're going to get some zeros.

0	1	0	0
16	25	36	49
64	0	0	0

- Or, with C99 syntax

```
int table[3][4] = {
  [ 1 ] = { 16, 25, 36, 49 },
  [ 0 ] = { [ 1 ] =  1 },
  [ 2 ][ 0 ] = 64
};
```

0	1	0	0
16	25	36	49
64	0	0	0

Multi-Dimensional Array Initialization

- You can let the compiler determine the array size
  - But just the outermost (first) dimension.

```
int a[ ][ 3 ] = { { 10, 15, 20 },
                  { 30, 35, 40 } };
```

OK

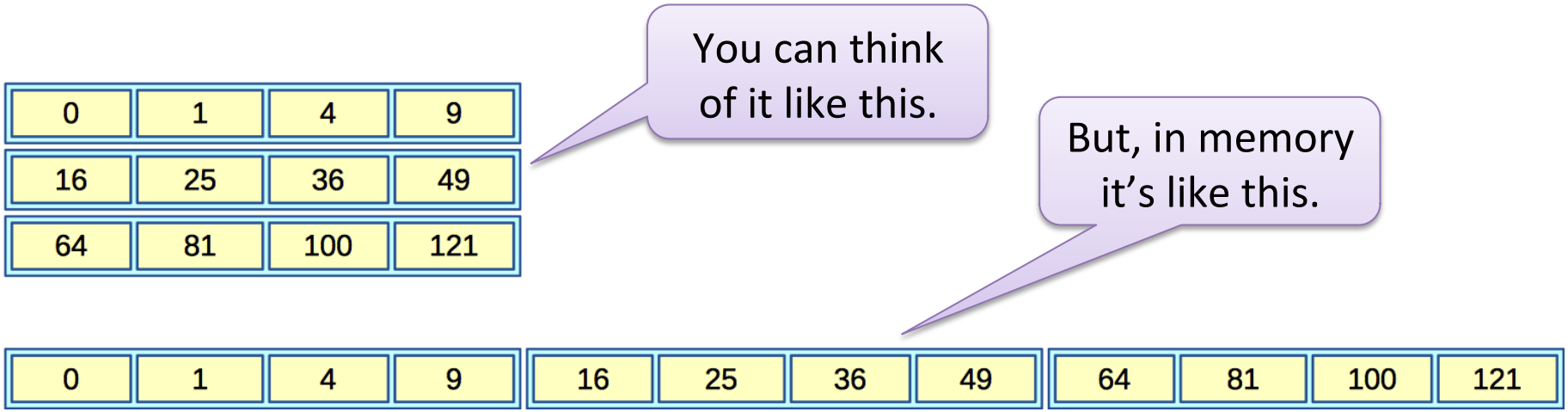
```
int a[ 2 ][ ] = { { 10, 15, 20 },
                  { 30, 35, 40 } };
```

Not OK

- Why the difference?
  - Size of all inner dimensions must be specified in the declaration.
  - It's used by the compiler to write the indexing code.

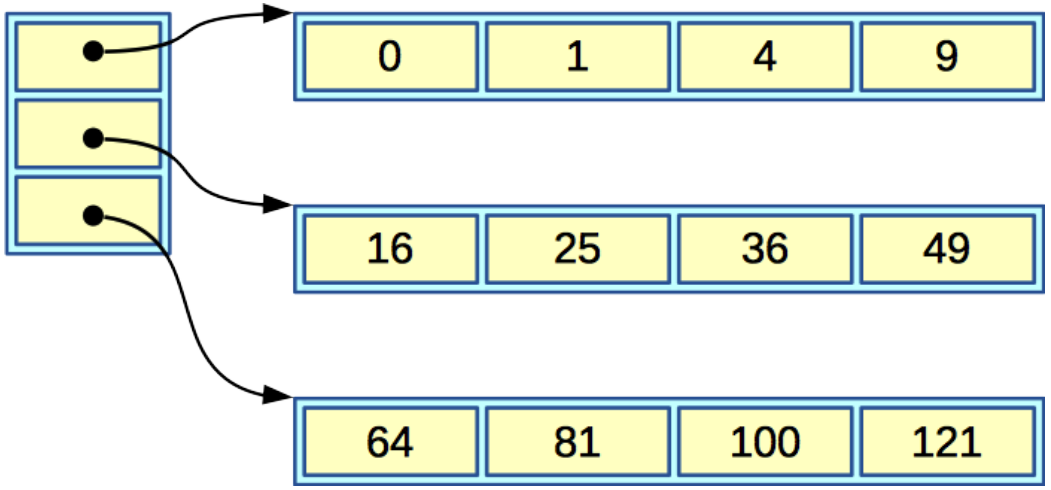
Memory Organization

- In memory, two-dimensional arrays are laid out in *row-major order*



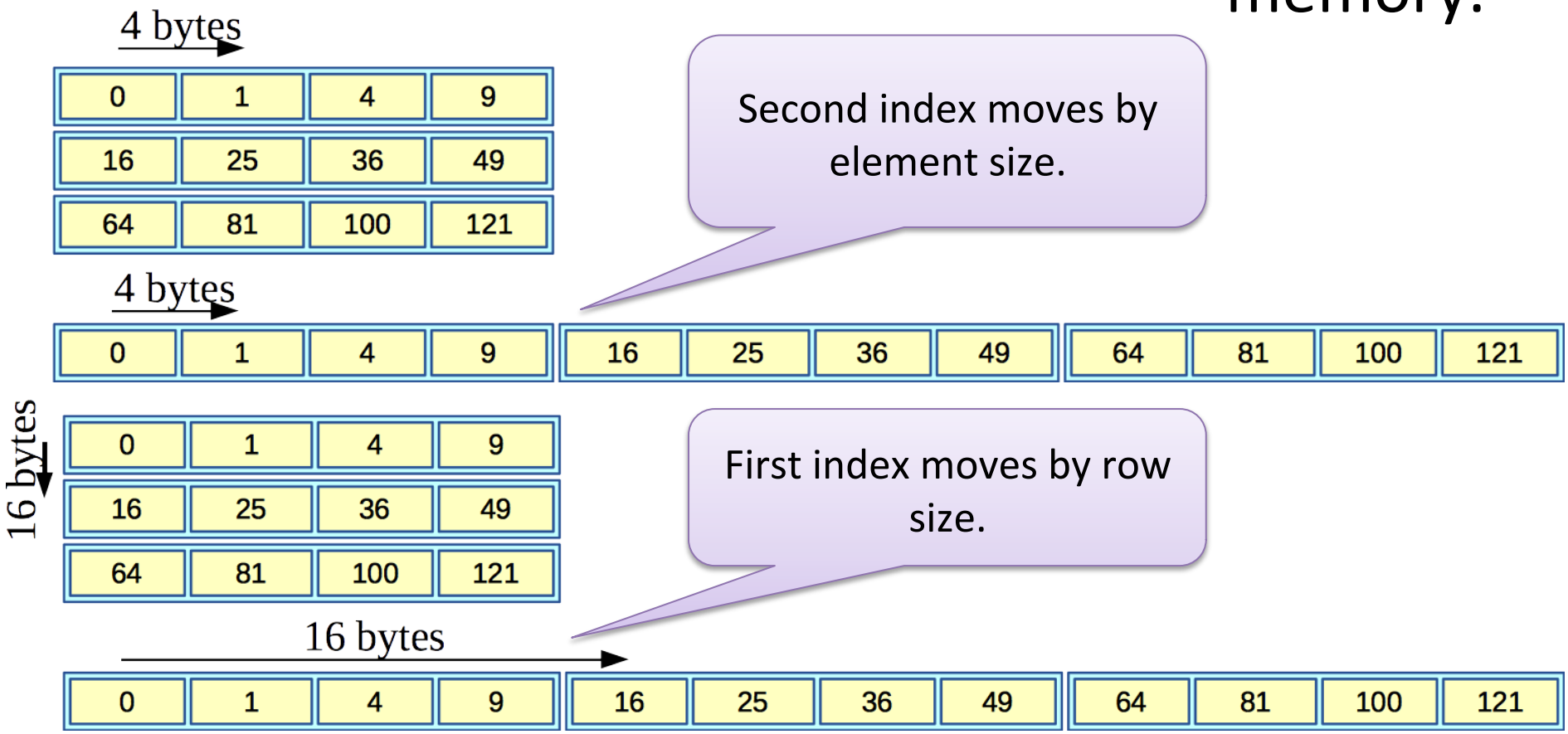
Memory Organization

- It’s not like this in Java.
  - It was more like this:



Memory Organization

- Fixed row size makes it easy to index into this memory.



Multi-Dimensional Arrays

- Multi-dimensional arrays are just really **arrays of arrays**.

```
int row[ 4 ];

int table[ 3 ][ 4 ];
```

I'm an array of 4 ints.

I'm an array of 3 guys like row.

- We can take an array apart, into its individual elements (rows)
  - .. and each row's elements (ints)

```
sizeof( table );
sizeof( table[ 0 ] );
sizeof( table[ 0 ][ 0 ] );
```

I'm the size of the whole array.

I'm the size of one of its elements, a row.

I'm the size of one of that row's elements.

Arrays and Stack Space

- Arrays can be stack-allocated.
- But, in practice, stack space is limited.
- A big array on the stack is an easy way to use it up.
- Or, deeply nested recursion will do it too.

```
int bill()
{
  int smallTable[ 10 ];

  bigTable[ 0 ] = 1;
  return bigTable[ 0 ];
}
```

I'm probably going to segfault.

```
int bill()
{
  int bigTable[ 100000000 ];

  bigTable[ 0 ] = 1;
  return bigTable[ 0 ];
}
```

Allocating Large Arrays

- Larger arrays can be allocated statically
  - The compiler can plan ahead for static storage.

The compiler will be sure to leave room for this.

```
int bigTable[ 100000000 ];

int bill()
{
  bigTable[ 0 ] = 1;
  return bigTable[ 0 ];
}
```

- Or a large array could be dynamically allocated … more about this later.

Arrays as Function Parameters

- An array can be passed as an function parameter
- The array parameter looks like an array declaration
- You can specify the array length explicitly in the function declaration … if you want.

```
void report( int table[8] )
{
    …
}
```

```
void report( int table[5][8] )
{
    …
}
```

Array parameter, with explicit size.

Array Parameter Dimensions

- In array passing, the compiler doesn't care so much about the outermost size.
- But it cares a lot about the inner sizes.

```
void report( int table[5][8] )
{
    …
}
```

```
int regularTable[5][8];
int    tallTable[6][8];
int    wideTable[5][9];
…
report( regularTable );
report( tallTable );
report( wideTable );
```

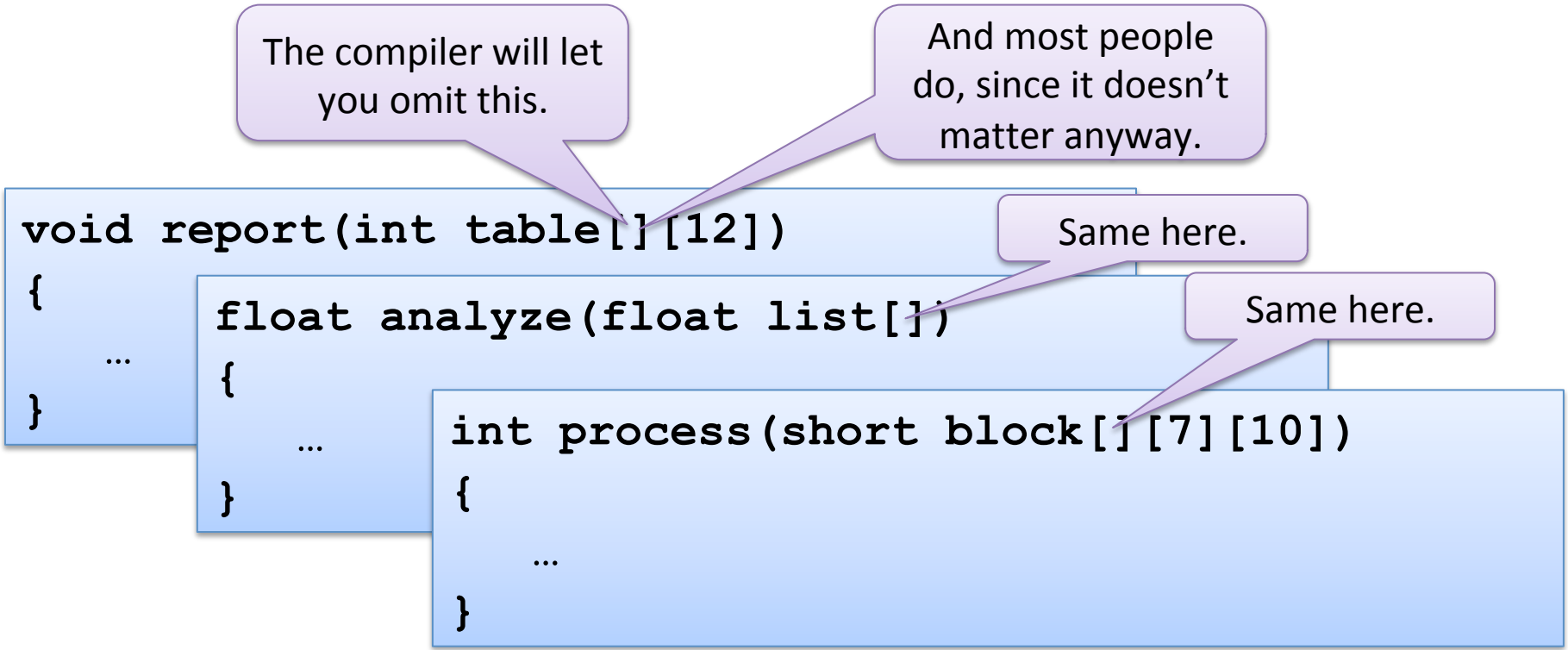
This will work

So will this.

But not this!

Array Parameter Dimensions

- In array indexing, the inner size determines the width of each row.
  - The compiler needs to know this to index between the rows..
- But, the outermost index just determines how far we go before we're out of bounds
  - And C doesn't do any bounds checking anyway.



Array Parameter Dimensions

- The function will work for any array length (at least for the outer index).
- It can't even tell how big the array is.

```
void report(int list[])
{
    int items = sizeof(list)/sizeof(list[0]));
}
```

This Doesn't Work!

Size of the whole array … divided by size of an element … should be number of elements.

But, the function doesn't know the size of the whole array.

This compiles, but it doesn't mean what you want it to.

Communicating Array Size

- How can a function tell how big the array is?
- Often, we need to pass that in as another parameter.

```
int countValues(int len, int list[])
{ … }
```

- For multi-dimensional, fixed-width arrays, this usually just applies to the outer-most dimension.

```
int countValues(int rows, int table[][7])
{ … }
```

- In some special cases, the function may be able to figure out how long the array is.

```
void countChars(char str[])
{ … }
```

Passing Variable-Sized Arrays

- For variable-sized arrays, you **must** pass everything but the outer-most dimension.

You need this just to pass the array.

See, it gets used here in the type.

```
void sumTable(int rows, int cols, int table[][cols])
{   …
    for ( i = 0 ; i < rows; i++)
        for ( j = 0; j < cols; j++)
            total += table[i][j];
    …
}
```

You may also need this to know how big the array is.

Passing Variable-Sized Arrays

```
void sumTable(int rows, int cols, int table[][cols])
{
   ...
}
```

- We depend on the caller to tell us the truth.

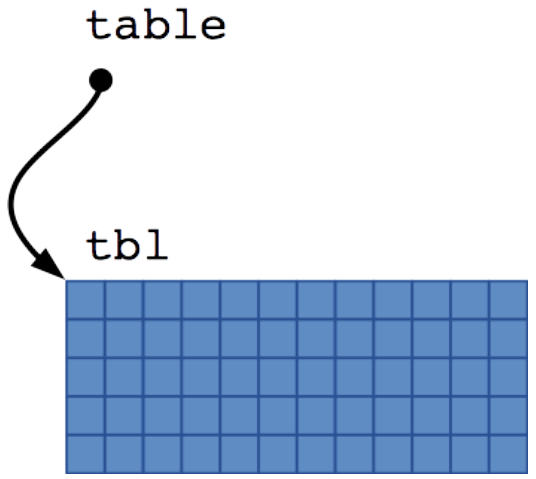
```
int tbl[10][12];

// Good client code.
sumTable( 10, 12, tbl );
// Evil client code.
sumTable( 20, 13, tbl );
```

Arrays as Parameters

- Secretly, arrays are automatically passed by reference (by starting address), not by value
  - This is why sizeof() doesn't do what you think it should for an array parameter

```
void useTable( int table[5][12] )
{
    size_t sz = sizeof( table );
}
```

```
int tbl[5][12] = { { … } };
. . .
useTable( tbl );
```

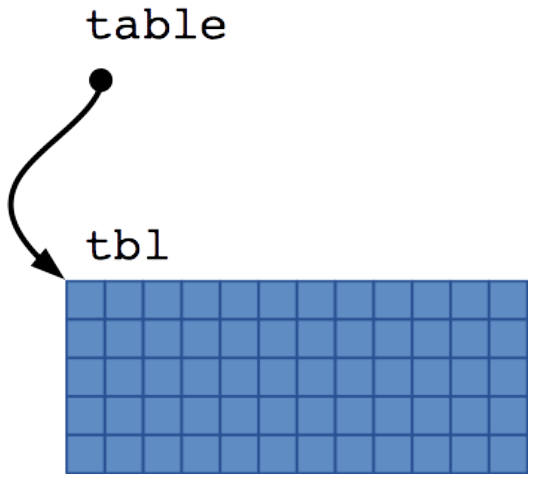


Arrays as Parameters

- So, the function can modify the contents of the caller's array

```
void changeTable( int table[5][12] )
{
   table[3][6] += 1;
}
```

```
int tbl[5][12] = { { … } };
. . .
changeTable( tbl );
```



Array Return Values

- Functions can have array parameters
- … but not array return values.

```
int makeList( int a, int b )[ 10 ]
{
    ...
}
```

I guess it would look like this, if you could do it … which you can't.

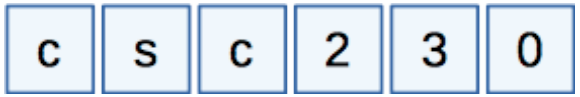
- Later, we'll figure out how to approximate this
  - by returning a pointer to an array.

Character Strings

- In C, a string is:
  - A sequence of characters stored in a one-dimensional array.
  - With a null character to mark the end.
- You can still create plain-old character arrays:

```
char a[] = { 'c', 's', 'c', '2', '3', '0' };
```

sizeof(a) will report 6.



- You also have a more convenient syntax for creating null-terminated strings:

```
char s[] = "csc230";
```

sizeof(s) will report 7.

Literal String Concatenation

- String concatenation works in the initialization syntax.

```
char s1[] = "Now " "is " "the " "time";
printf( "%zd\n", sizeof( s1 ) );
```

I'm 16 bytes.

- But, you can only use this on literal strings.
  - It doesn't do general-purpose string concatenation.

```
char s3[] = s s1;
```

This won't work.

Strings in C

- Null termination is a convention in C
  - It lets us mark the end of a string in memory, without explicitly storing the length.
  - Standard string processing functions expect this.
  - For example, **`printf()`** can print a string with **`%s`**
  - … provided it's null terminated.

```
char word[] = "csc203";
printf( "The string is %s\n", word );
```

Reading Strings

- scanf() will read strings with %s
  - It reads a single word (a space-delimited sequence of non-whitespace characters)
- Scanf follows the convention of null-terminating the string it reads.