Be a Pointer Expert

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics For Today

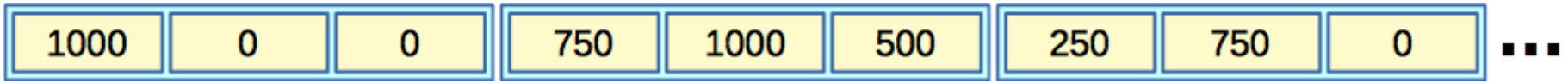
- Pointers and Multi-Dimensional Arrays
  - Pointer types for 2D arrays
  - Dynamically allocating 2D arrays
- 2D arrays via arrays of pointers
  - Dynamically allocating these
- Arrays of Strings
  - As 2-dimensional arrays
  - As arrays of char pointers
- Command-Line Arguments
- Pointers to Functions

Multi-Dimensional Arrays

- Remember, a 2-dimensional array is just an array of arrays.
- Like this one. It's a 6-element array of 3-element arrays of shorts.

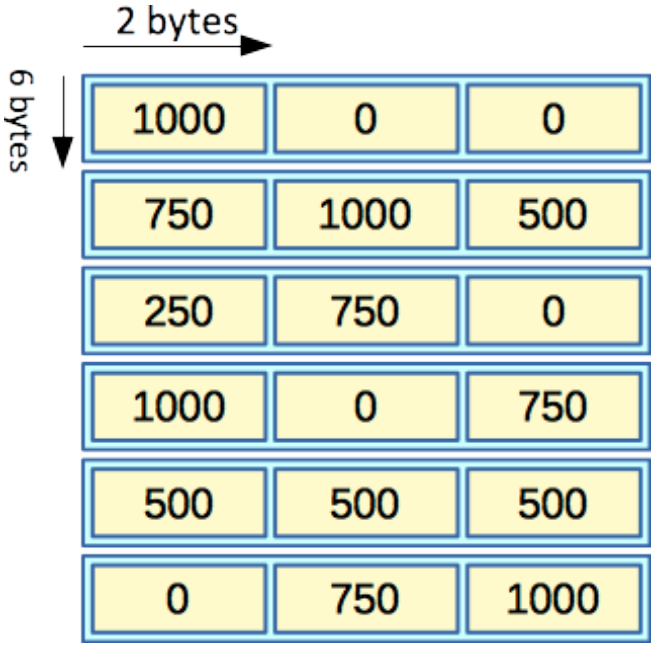
```
short palette[][ 3 ] = {
    { 1000, 0, 0 },
    { 750, 1000, 500 },
    { 250, 750, 0 },
    { 1000, 0, 750 },
    { 500, 500, 500 },
    { 0, 750, 1000 }
};
```

- In memory, we can think of this as building something like this.



Thinking about 2D Arrays

- But, we probably like to think of it more like this:
- The important part:
  - With each row the same length, the compiler knows where to find each element based on its indices.

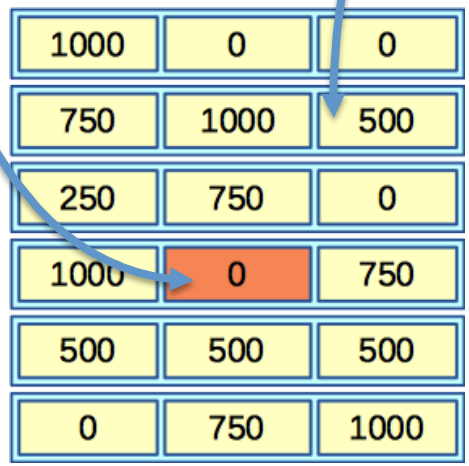


Taking Multi-Dimensional Arrays Apart

- There are lots of different ways to access the contents of this array.
- Just a plain old value: `palette[1][2]`
- Address of a value: `&(palette[3][1])`

Do we need these parentheses?

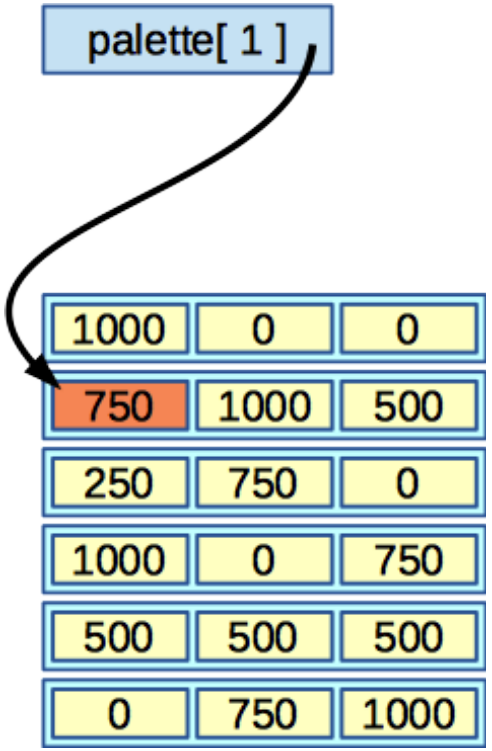
```
short palette[][ 3 ] = {
    { 1000, 0, 0 },
    { 750, 1000, 500 },
    { 250, 750, 0 },
    { 1000, 0, 750 },
    { 500, 500, 500 },
    { 0, 750, 1000 }
};
```



Taking Multi-Dimensional Arrays Apart

- Each row is a 3-element array.
- So, an expression like: `palette[i]`
  - Should evaluate to row i, an array of shorts
  - And it does …
  - Like any other array, it evaluates to the address of the row's first element.
  - So, palette[1] has type "pointer to short"

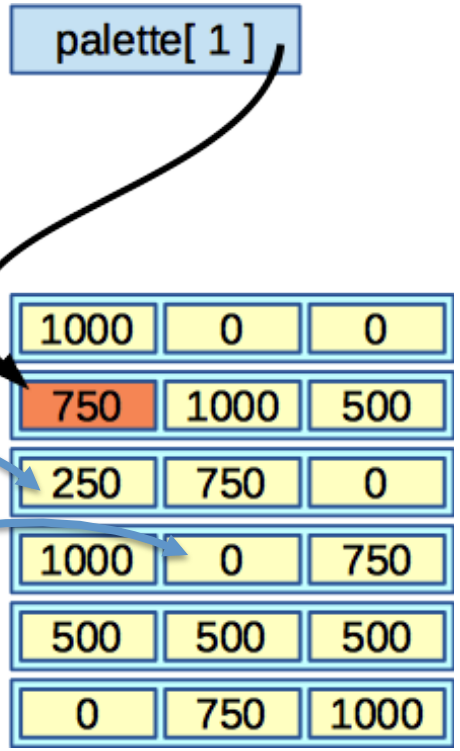
```
short palette[][ 3 ] = {
    { 1000, 0, 0 },
    { 750, 1000, 500 },
    { 250, 750, 0 },
    { 1000, 0, 750 },
    { 500, 500, 500 },
    { 0, 750, 1000 }
};
```



Taking Multi-Dimensional Arrays Apart

- We can use palette[1] like any other pointer to short.
  - Copy it to a pointer variable.
  - Pass it to a function.
  - Do pointer arithmetic with it.

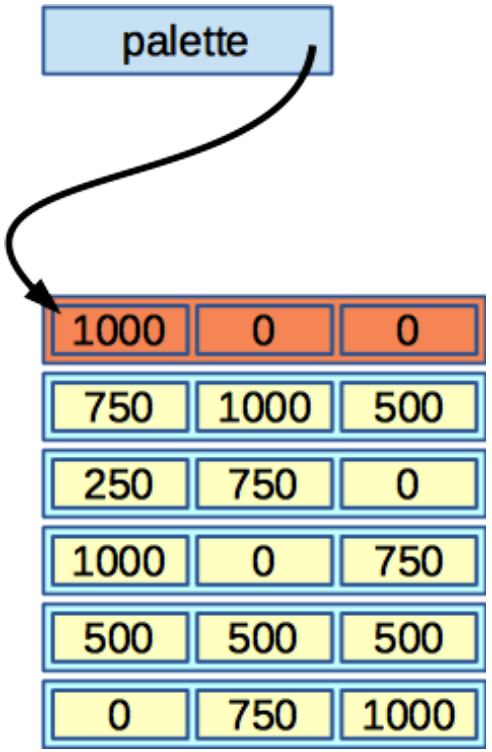
```
short palette[][ 3 ] = {
    ...;
};

short *p = palette[ 2 ];
short *q = palette[ 3 ] + 1;

*q = 50;
```



Taking Multi-Dimensional Arrays Apart

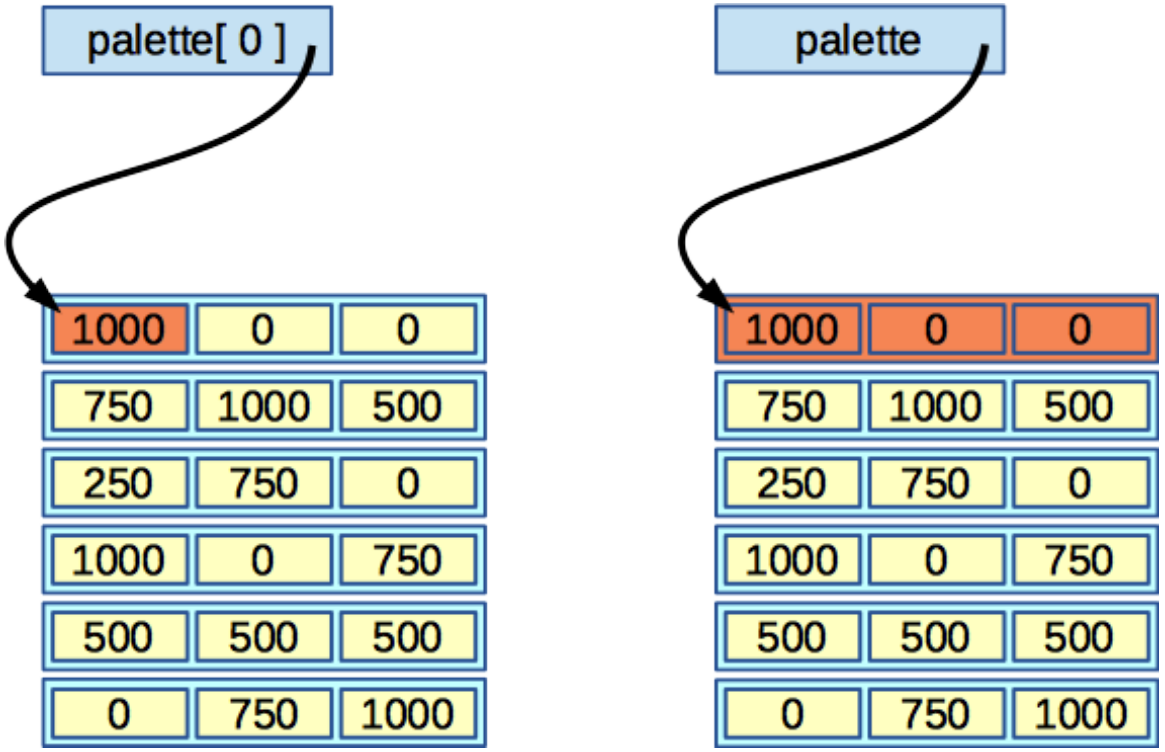
- So, syntax like palette[i] lets us work with rows
  - via pointers to a row's first element.
- What about syntax like: palette
  - This must be a pointer that lets us get to the whole 2D array.
  - i.e., a pointer to its first element, the first row.

```
short palette[][ 3 ] = {
    { 1000, 0, 0 },
    { 750, 1000, 500 },
    { 250, 750, 0 },
    { 1000, 0, 750 },
    { 500, 500, 500 },
    { 0, 750, 1000 }
};
```



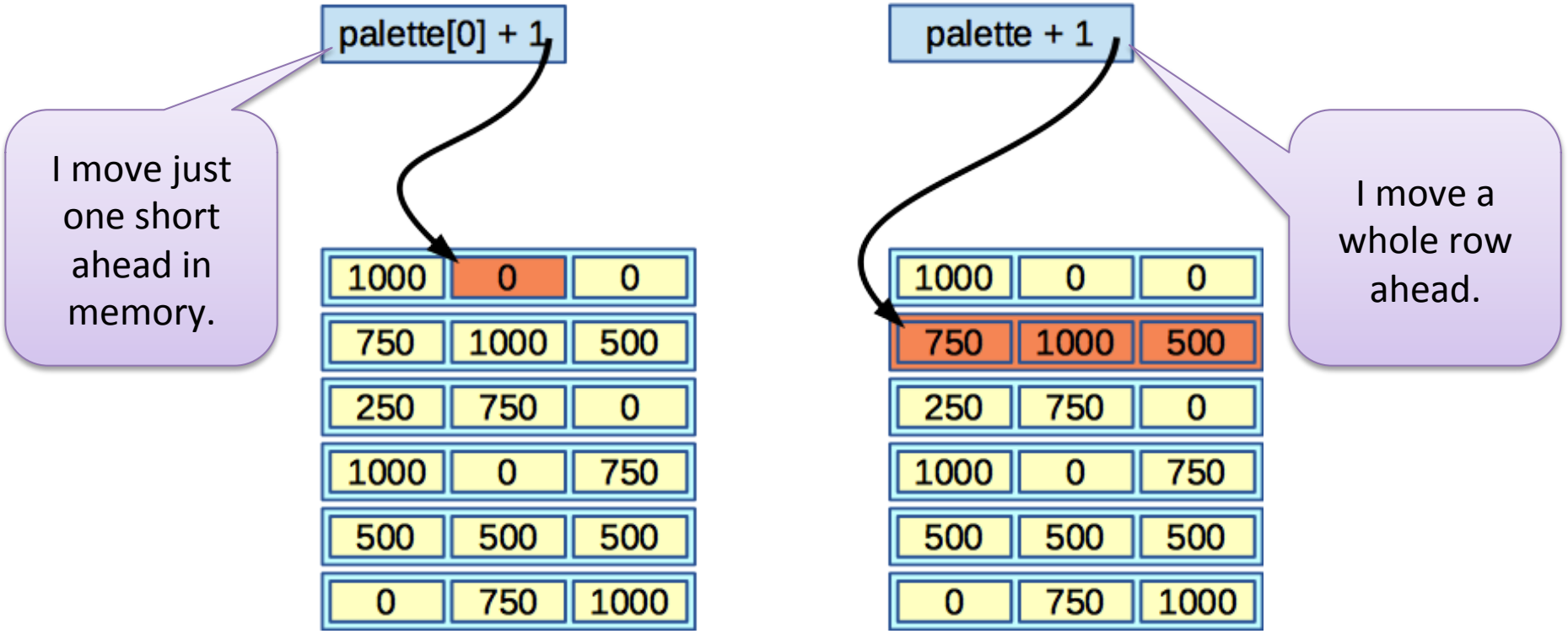
Comparing Pointers

- What's the difference?
  - These pointers point to different-sized things.

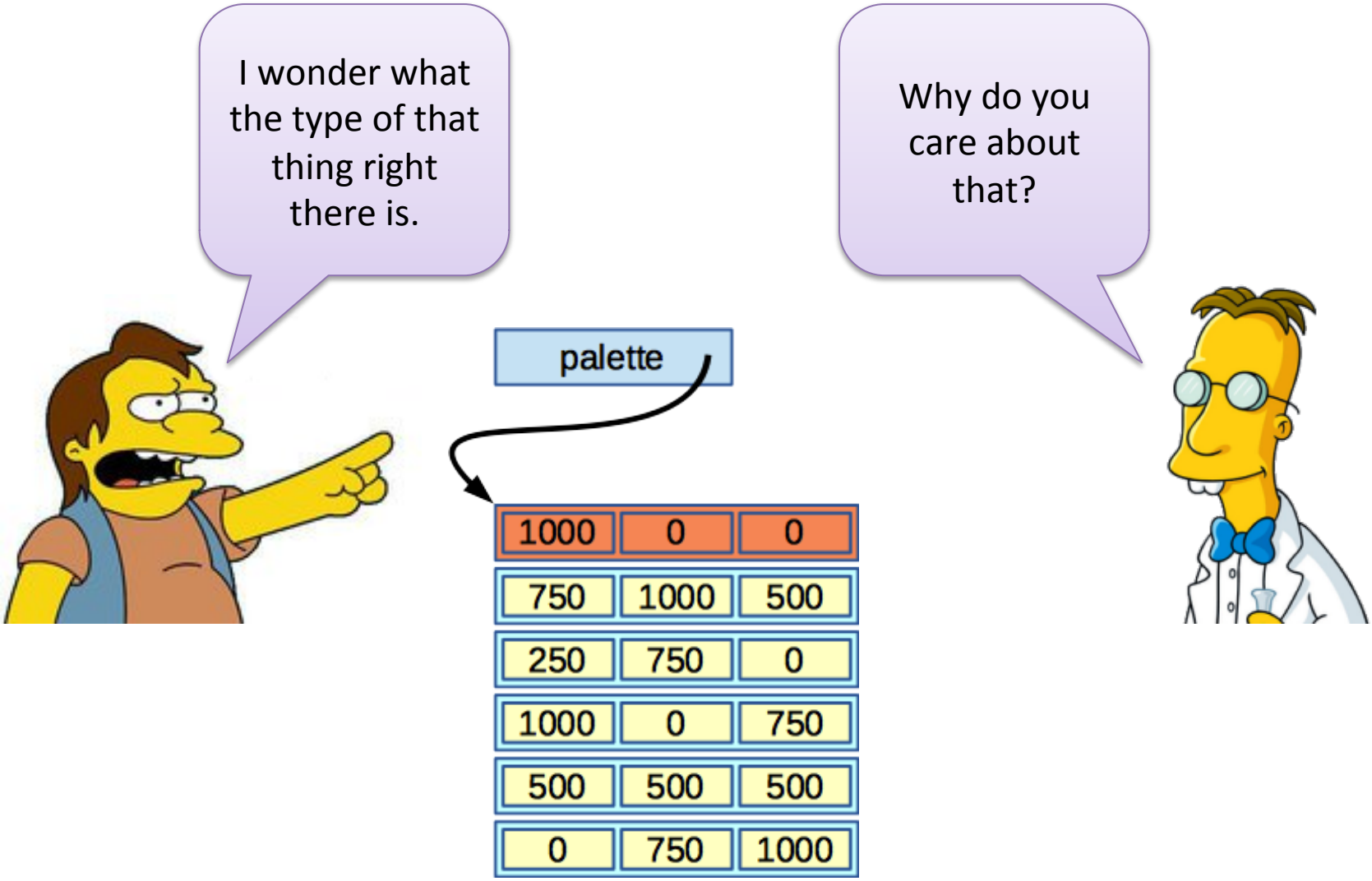


Type Matters

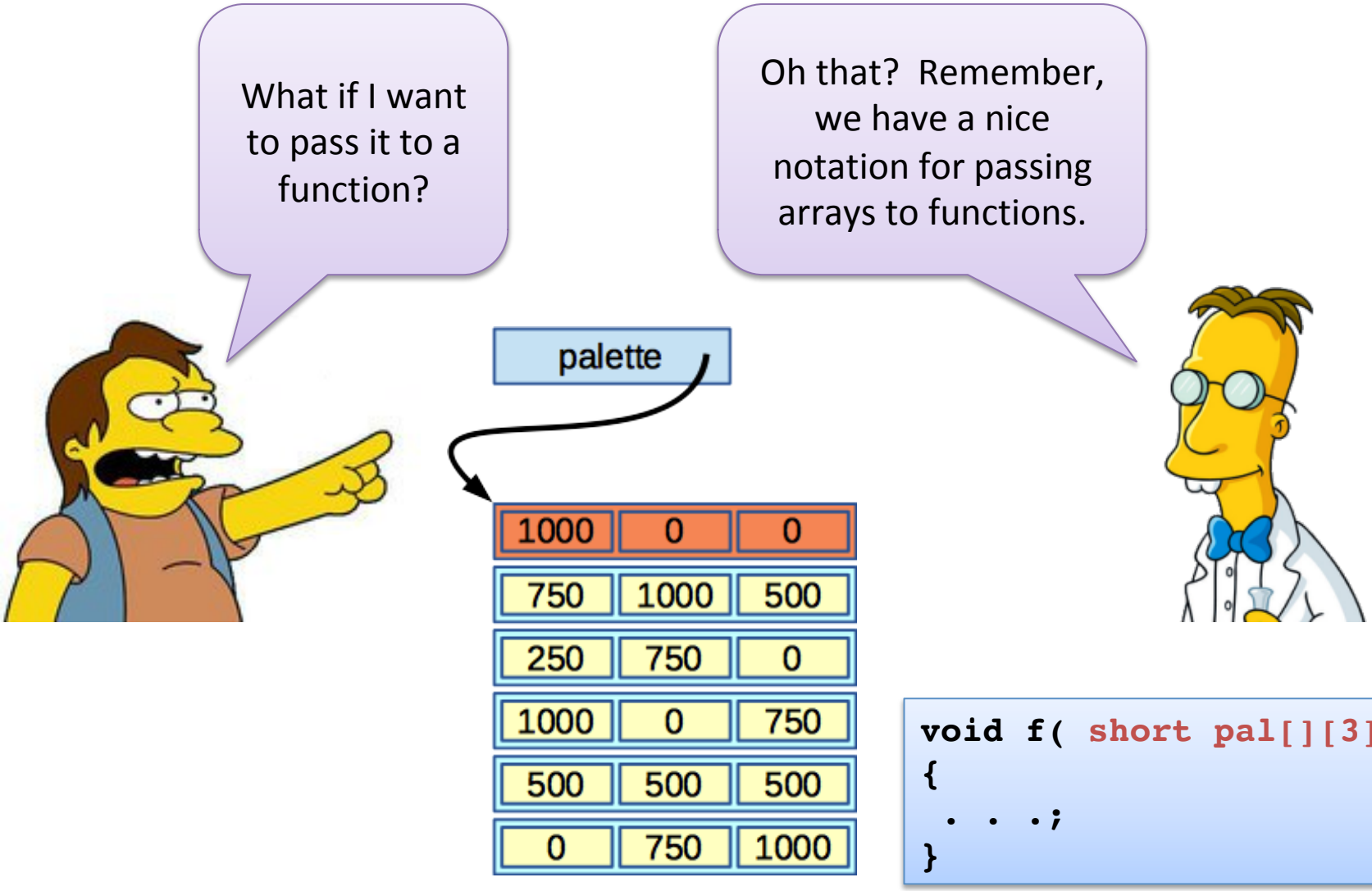
- Since they point to different sized things.
  - They behave differently under pointer arithmetic.



Type Matters

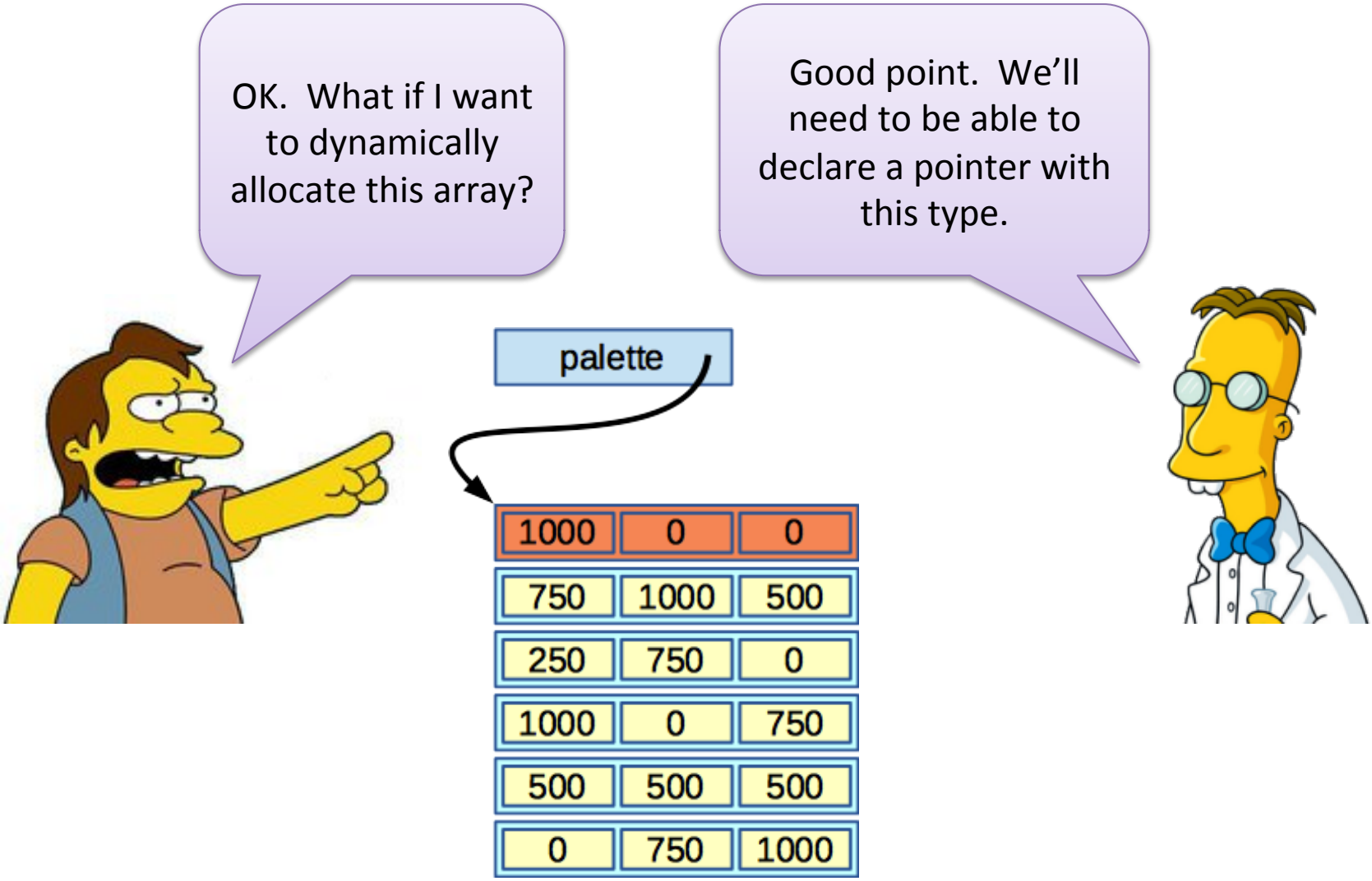


Type Matters




```
void f( short pal[][3] )
{
  . . .;
}
```

Type Matters



Taking Multi-Dimensional Arrays Apart

- Well, if **A** is an array of **X**s,
  A evaluates to a pointer to an **X**
- palette is an array of colors
  (where a color is an array of 3 shorts)
- So, palette evaluates to a pointer to a 3-element array of short.
- How do we write this?
  - short color[3];
  - short (*cptr)[3];

I'm an array of three shorts.

I'm a pointer to an array of three shorts.

Do we need these parentheses?

```
short palette[][ 3 ] = {
    { 1000, 0, 0 },
    { 750, 1000, 500 },
    { 250, 750, 0 },
    { 1000, 0, 750 },
    { 500, 500, 500 },
    { 0, 750, 1000 }
};

short (*cptr)[3];
cptr = palette;
```

Taking Multi-Dimensional Arrays Apart

- We can use this type to pass palette to a function.

```
void f( short (*p)[ 3 ] ) {
  ...;
}
```

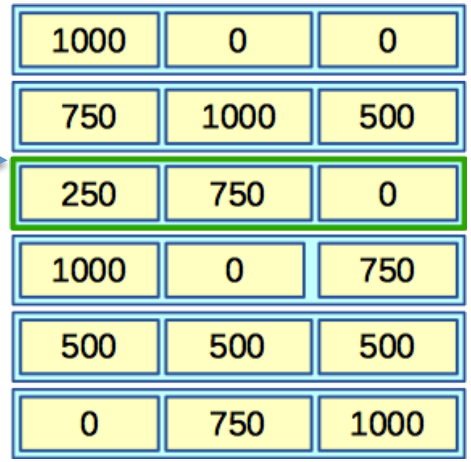
```
f( palette );
```

- We could get an offset into the palette array.

```
short (*p2)[ 3 ] = palette + 2;
p2[ 1 ][ 2 ] = 25;
```

- Or pass that to a function.

```
f( palette + 3 );
```



Taking Multi-Dimensional Arrays Apart

- We have an alternative, simplified syntax for passing arrays to functions.

```
void f( short p[][ 3 ] ) {
  ...;
}
```

```
void f( short (*p)[ 3 ] ) {
  ...;
}
```

- And this is what it means.
- Elsewhere, we have to use the pointer syntax.
- Let's practice.

```
int a[ 5 ][ 10 ];
int (*ap)[ 10 ] = a;

double b[ 10 ][ 5 ][ 3 ];
double (*bp)[ 5 ][ 3 ] = b;
```

I'm an array of 5 things.

I'm a pointer to the first one.

I'm an array of 10 things.

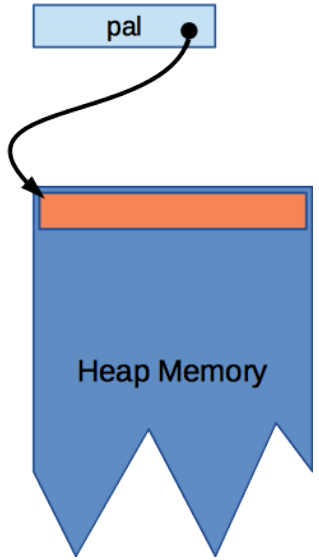
I'm a pointer to the first one.

Multi-Dimensional Arrays

- These pointer types are exactly what we need to dynamically allocate 2D arrays.

```
short (*pal)[ 3 ];
int N = 1000;
pal = (short (*)[3])malloc(N * 3 * sizeof(short));
```

There's something a little goofy here



- Now, we can use this pointer just like an N X 3 array of shorts.

```
pal[ 5 ][ 2 ] = 850;
```

Multi-Dimensional Arrays

- C99 lets us define variable-sized arrays
- The same technique lets us allocate variable-sized arrays on the **heap** instead of the stack.

```
int R = 500;
int C = 400;

int (*a)[ C ];

a = (int (*)[C])malloc( R * C * sizeof( int ) );

for ( int i = 0; i < R; i++ )
  for ( int j = 0; j < C; j++ )
    a[ i ][ j ] = i * j;
```

I'm a pointer to a row of C ints.

Now, I point to the first of R rows.

See, works just like a 2D array.

Freeing Memory

- The last two arrays are each stored in one big block.
- We can free them with one call to free().

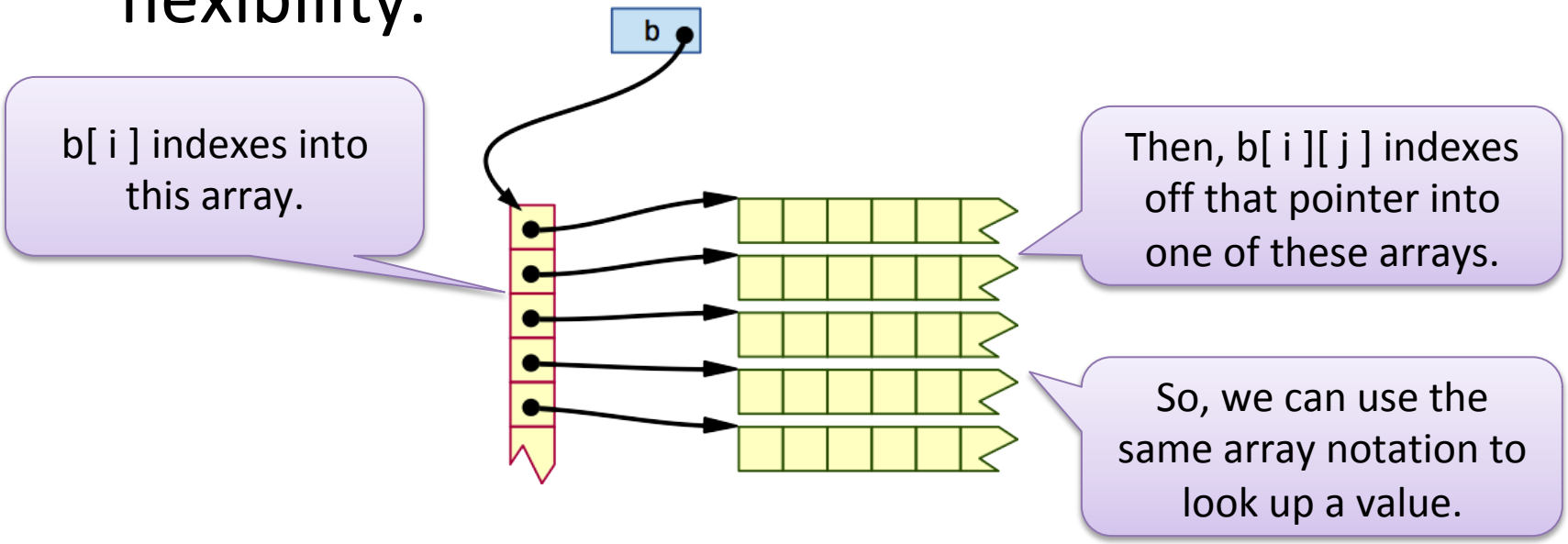
```
free( pal );
free( a );
```

Free the N X 3 array of shorts.

Free the R X C array of ints.

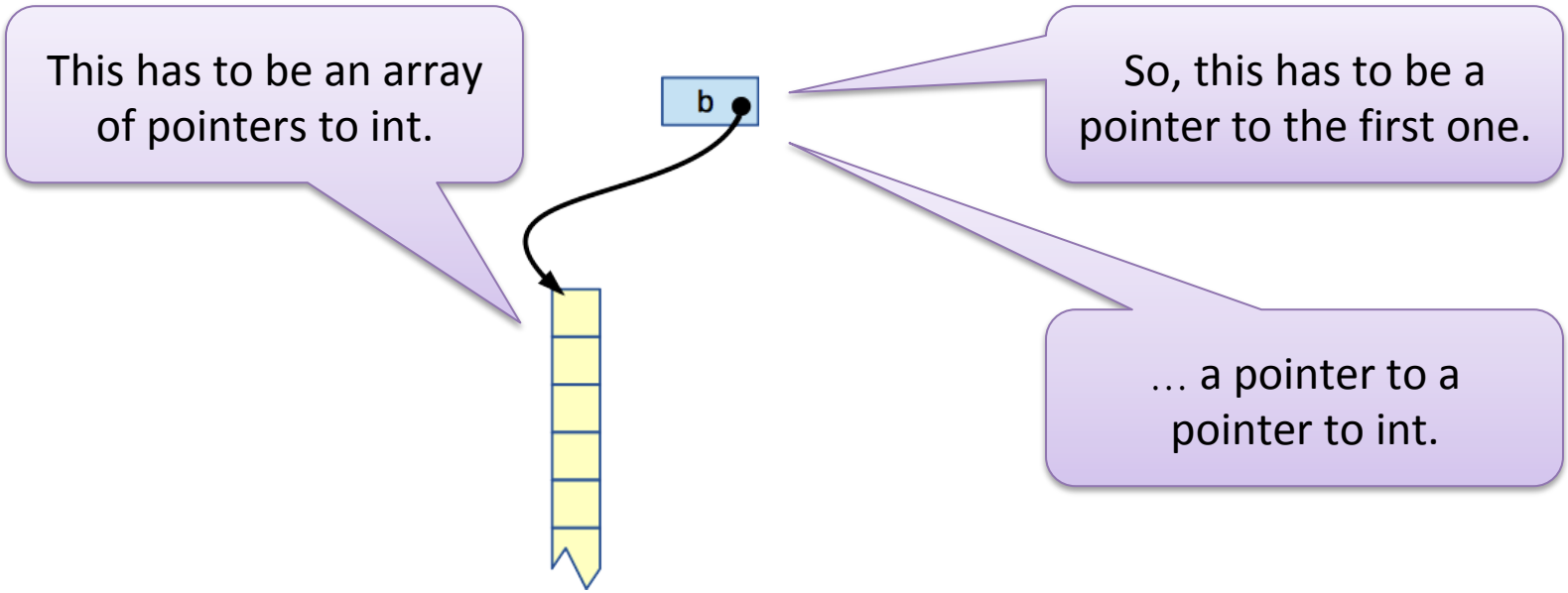
Arrays of Pointers

- Alternatively, you can structure your array Java-style, as an array of pointers to arrays.
- This takes some more work, but it offers more flexibility.



Arrays of Pointers

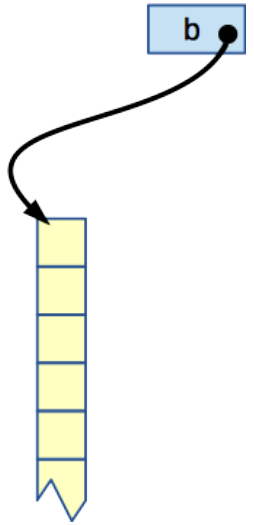
- We can figure out what types we need to make this work.



Arrays of Pointers

- First, we have to create the outer array

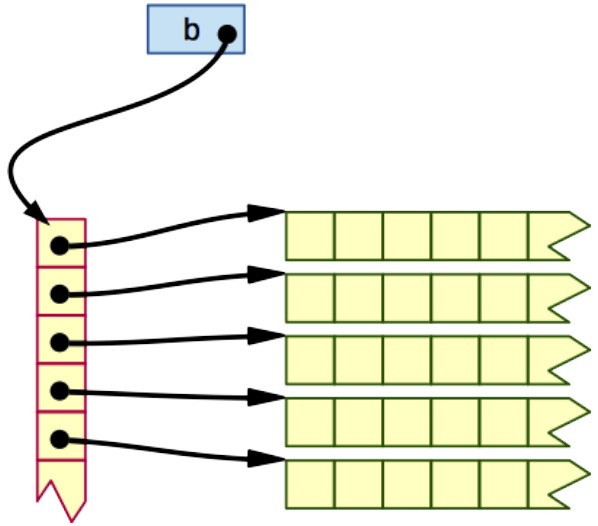
```
int **b;
b = (int **)malloc( R * sizeof( int * ) );
```



Arrays of Pointers

- First, we have to create the outer array …
- … then each of the inner arrays.

```
int **b;
b = (int **)malloc( R * sizeof( int * ) );

for ( int i = 0; i < R; i++ )
    b[ i ] = (int *)malloc( C * sizeof( int ) );
```

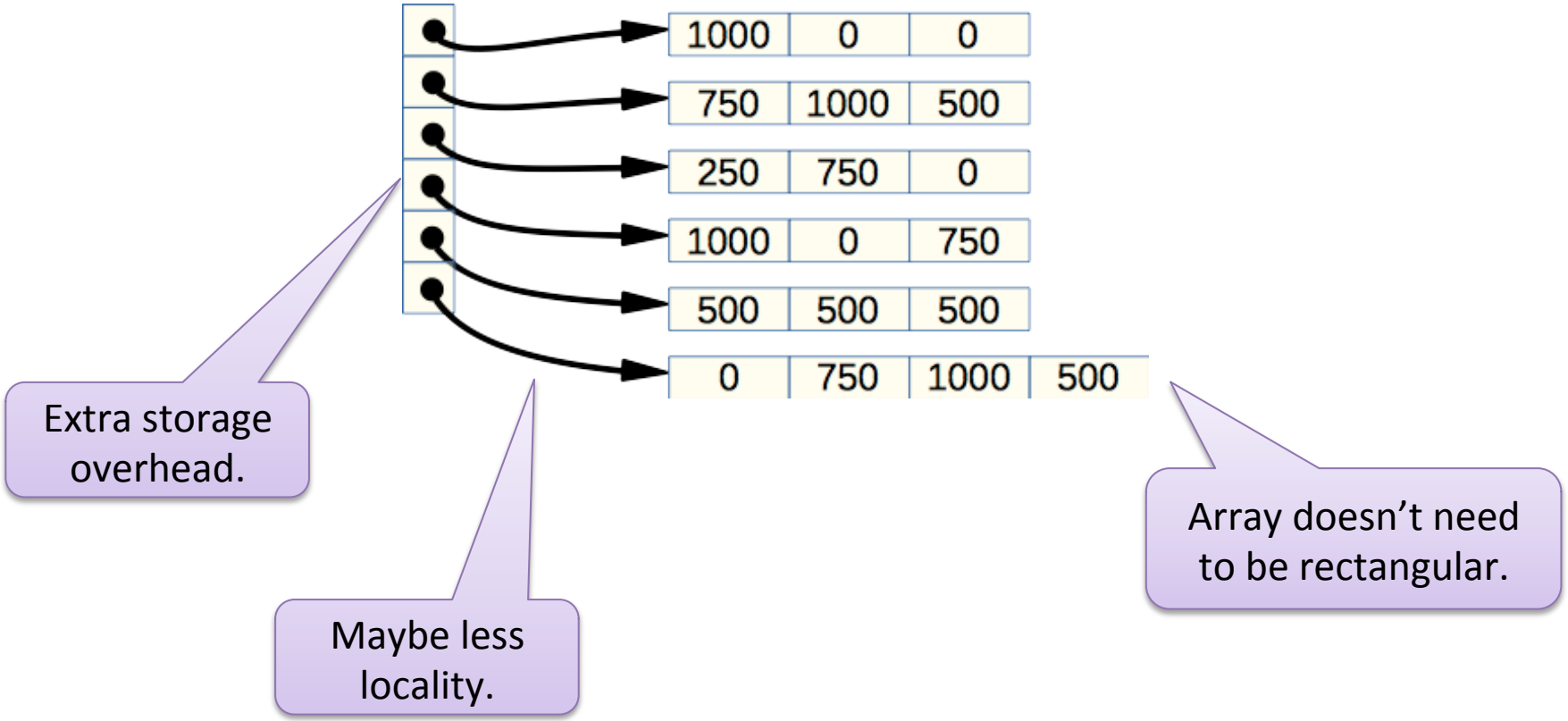


- When it's time to free the array, it's going to take more work.

```
  for ( int i = 0; i < R; i++ )
    free( b[ i ] );
  free( b );
```

Arrays Of Pointers

- There are some tradeoffs with this representation.



Arrays Of Strings

- We can use similar techniques for representing arrays of strings.

A 2D Array of characters.

```
char words[][ 14 ] = {
    "eggplant",
    "nap",
    "establishment",
    "synergy",
    "paleolithic",
    "seaweed"
  };
```

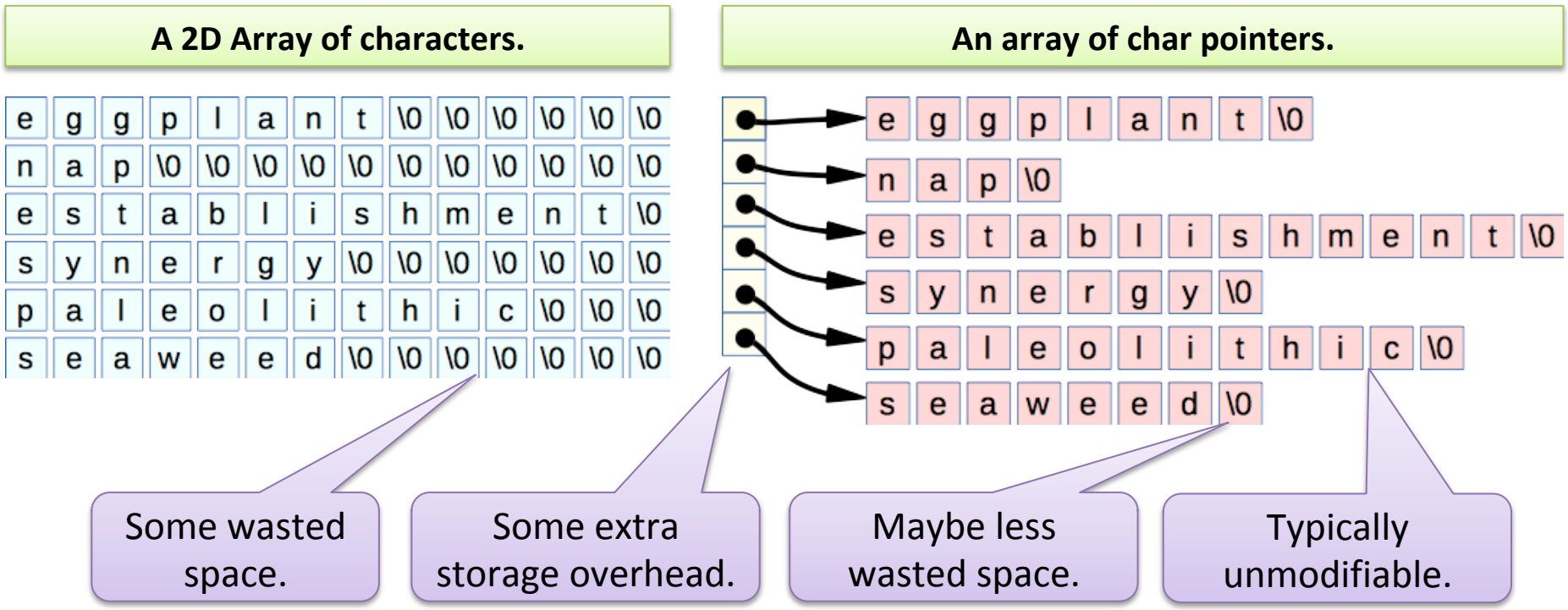
An array of char pointers.

```
char *words[] = {
    "eggplant",
    "nap",
    "establishment",
    "synergy",
    "paleolithic",
    "seaweed"
  };
```

Here, we're doing stack allocation instead of heap.

Arrays Of Strings

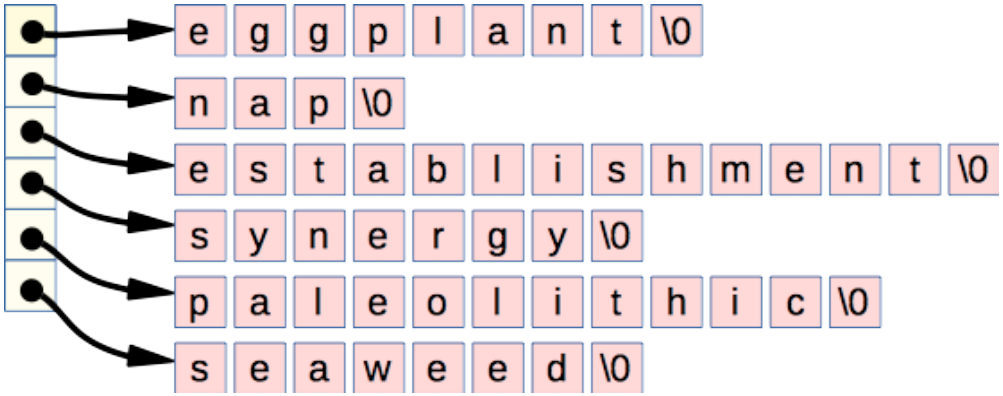
- Here also, there are tradeoffs.



Arrays Of Strings

A 2D Array of characters.

An array of char pointers.

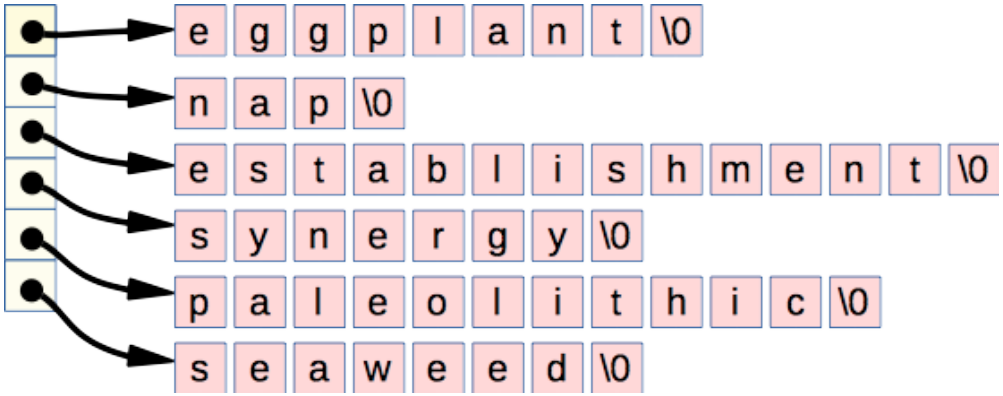


- Here words has type: char (*ptr)[14];
- But here, it has type: char **ptr;
- for both:
  - words[3][5] is a character (g)
  - words[1] is a char pointer (nap)
  - words gets you the whole array (but the types differ)

Arrays Of Strings

A 2D Array of characters.

An array of char pointers.



- What will sizeof() say for each of these types?

Command Line Arguments

- Main can take parameters that give access to the command-line arguments.

```
int main( int argc, char *argv[] ) …
```

Number of arguments.

Array of char pointers, one for each argument

- When you run your program, you decide what these arguments will be.

```
shell$ ./myProgram a b "c d e"
```

I'm argument 0

I'm argument 1

I'm argument 2

I'm argument 3

Command Line Arguments

- Say, we wanted to just report the arguments:

```
int main( int argc, char *argv[] )
{
  printf( “%d arguments\n", argc );

  for ( int i = 0; i < argc; i++ )
    printf( "    %s\n", argv[ i ] );
}
```

- More typically, we would use the arguments to change how the program operates.

Pointers to Functions

- There's anther type of pointer, a pointer to a function.
- We can store a pointer for the starting address of a function, and call the function later via this pointer
  - If this is a function:
    `int f( char *str )` …
  - This is a pointer to a function:
    `int (*fp)( char * );`

This can hold a pointer to function f, or **any** function of the same type.

We're declaring a variable named fp.

This looks like a function prototype, with all the parameter types.

… and, a star and parentheses around the function name.

Storing Function Pointers

- The name of a function evaluates to a pointer to that function.
- So, we can store a function pointer like this:

```
int myFunction( char *str ) {
  …;
  return someValue;
}
```

```
int (*fp)( char *str );
fp = myFunction;
```

```
int x = fp( “hello world” );
```

Here, we copy the function’s starting address to a function pointer.

Here, we call that function via a pointer.

Using Function Pointers

```
int stringLength( char *str )
{
  char *p = str;
  while ( *p )
    p++;
  return p - str;
}
```

```
int main( int argc, char *argv[] )
{
  int (*f)( char * );

  f = stringLength;

  for ( int i = 0; i < argc; i++ ) {
    int result = f( argv[ i ] );
    printf( "%d\n", result );
  }
  …
```

Print the length of each string on the command line.

This is worth something, but it's not how you normally use function pointers

Using Function Pointers

```
int applyFunction( char **strings,
                   int stringCount,
                   int (*f)( char * ) )
{
  int total = 0;
  for ( int i = 0; i < stringCount; i++ ) {
    total += f( strings[ i ] );
  }
  return total;
}
```

This is more typical.
This will apply any function f to the list of strings, returning the total.

```
  applyFunction( argv, argc, stringLength );

  applyFunction( argv, argc, countVowels );
```

We can get applyFunction() to use different functions, without even knowing the particular function it's using.

Reading Types

- Now, we have lots of building blocks for defining new types
  - fundamental and specialized types
  - arrays, pointers and function pointers
  - const
  - And we're not done yet
- We have a whole syntax for defining *type expressions*
- But, we still just need the same rules for reading type
  - Start at the name of the thing being declared (or where its name would be if it had a name)
  - Work your way out, respecting precedence

Fun With Reading Types

- Can you describe the types of the following?

```
int (*f)( int, void (*)( char ) );
```

```
void (*(*g)( int const *, float ))();
```

```
int *(*h)( float const * const *,
           double (*)( float ) );
```

- This isn't so bad. Even Wikipedia says: "An example of a simple type system is that of the C language."

A Larger Example

- I have listOfFunctions.c as a larger example … let's take a look.
  - It maintains a list of function pointers.
  - Gives us some practice reading and writing function pointer types

```
bool (*testList[ 10 ])( int );
```

  - Demonstrates the *chain of responsibility* pattern.

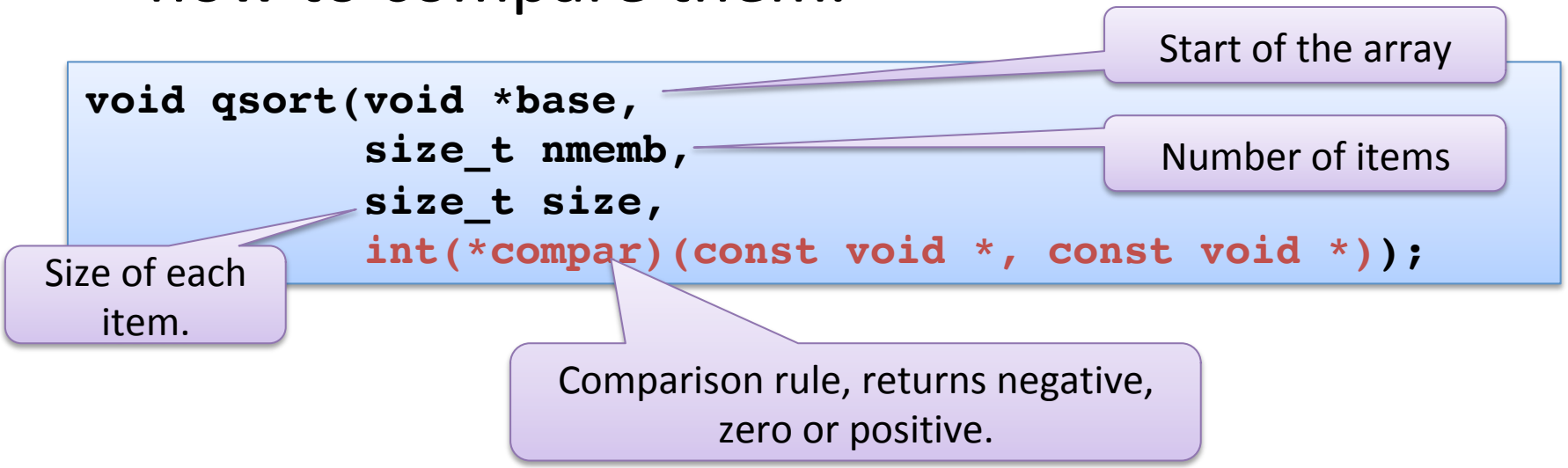
Function Pointers in the Standard Library

- Function pointers have been around since the beginning.
- Consider atexit(). It lets you register a function to be called at normal program termination.

```
int atexit( void (*function)(void) );
```

Function Pointers in the Standard Library

- There's also qsort(). It lets you sort arrays of anything you want. You just have to tell it how to compare them.



- The pointersInLibc.c example demonstrates this and atexit()

Sort Example

- Pretend we want to sort an array of ints.

```
int list[] = {
  25,
  10,
  19,
  32,
  7
};

qsort( list, 5, sizeof( list[ 0 ] ), intComp );
```

I say I take two void pointers, but, inside, I really expect two int pointers.

Sort Example

- We have to write a function that claims to take two void pointers.

```
int intComp( const void *aptr, const void *bptr )
{
  int const *a = aptr;
  int const *b = bptr;

  if ( *a < *b )
    return -1;
  if ( *a > *b )
    return 1;
  return 0;
}
```

But, inside, we know we're really getting two int pointers.

Thinking About Function Pointers

- You should think of function pointers the same way you think about abstraction and overriding in Java
  - We can talk about something (a function) and store it
  - … without committing to exactly what it will do when we use it.
- Before C++, this was an important part of how object-oriented code was written in C.