C-Programming Review Pointers & Arrays

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

C-Programming Review

POINTERS

Review of pointers

- What is a Pointer?
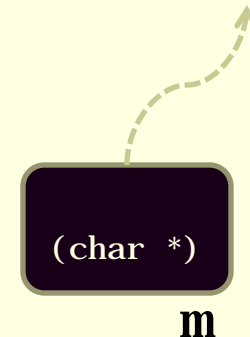
- # An Address!

Review of pointers

- A pointer is just a memory location.
  - In other words, a pointer is a <u>data type</u> whose **value** refers directly to ("points to") another value stored in computer memory

- The contents at a particular memory location are just a collection of bits – there's nothing special about them that makes them `ints`, `chars`, etc.
  - How you want to interpret the bits is up to you.
  - Here's a memory locaiton: Is this... an `int` value?
  ... a pointer to a memory address?
  ... a series of `char` values?

`0xfe4a10c5`

Review of pointer variables

- A pointer variable is just a variable, that contains a value that we **interpret** as a memory **address**.

- Tidbit:

  - Just like an uninitialized int variable holds some arbitrary "garbage" value,

  - an uninitialized pointer variable points to some arbitrary "garbage address"

  - Given this declaration:

    - `char *m;`

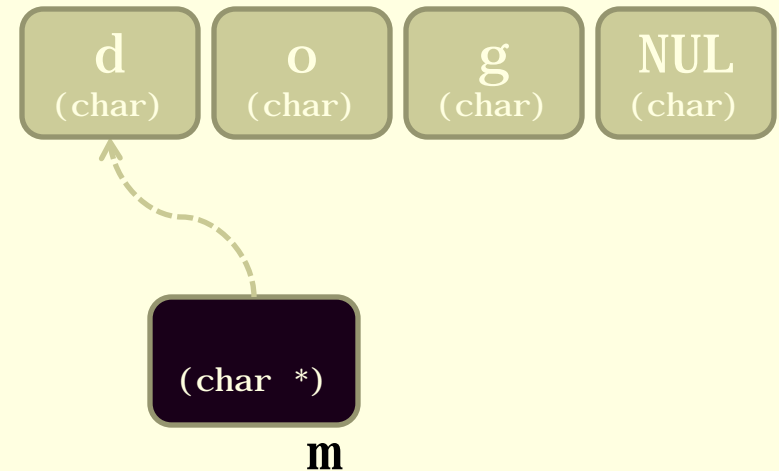   - we get an uninitialized pointer that points to some "garbage" value

(char *)

m

Indirection operator *

■ Moves from address to contents

`char    *m = "dog";`

d (char)	o (char)	g (char)	NUL (char)

(char *)

m

■ The * is a dereferencing operator

■ "dog" is copied to the address pointed to by m

Indirection operator *

- **Moves from address to contents**

```
char   *m = "dog";

char result = *m;
```

| d (char) | o (char) | g (char) | NUL (char) |

(char *)

m

d (char)
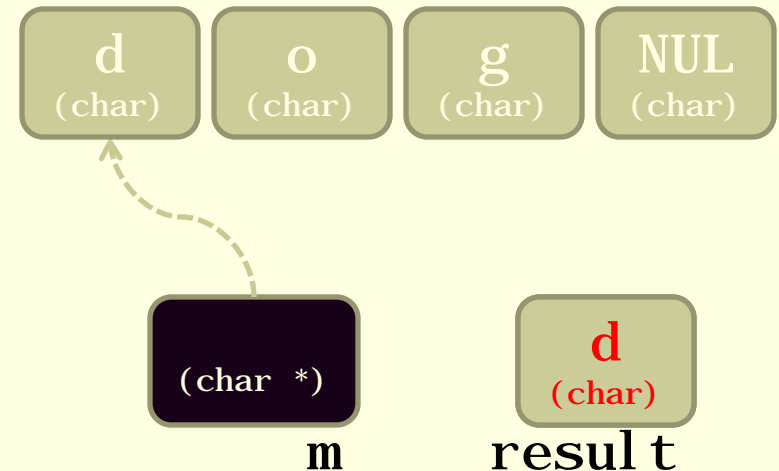
result

m gives an address of a `char`

`*m` instructs us to take the contents of that address

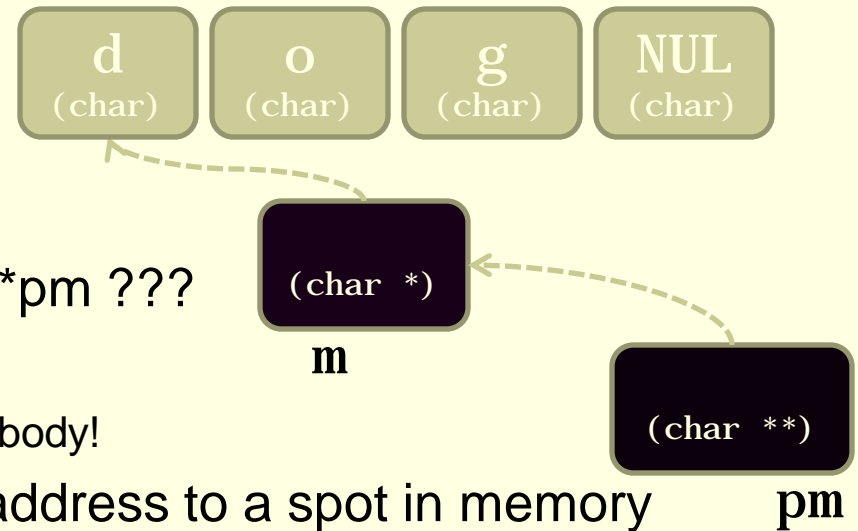`result` gets the value `'d'`

Address operator **&**

- ## Instead of contents, returns the address

```
char   *m = "dog",
       **pm = &m;
```

d (char)	o (char)	g (char)	NUL (char)

(char *)

**m**

(char **)

pm

- First of all, what the heck is the **pm ???
- This is a double pointer!
  - Yeah, I know, you wanna slap somebody!
- Remember, a pointer stores an address to a spot in memory
- Similarly, a double pointer stores an address to a spot in memory, and then that spot in memory also stores an address to another spot in memory
  - Which in this case is to a spot in memory that supposedly stores a char

Address operator **&**

■ Instead of contents, returns the address

```
char    *m  =  "dog",
        **pm  =  &m;
```

| d (char) | o (char) | g (char) | NUL (char) |

(char *)

m

(char **)

pm
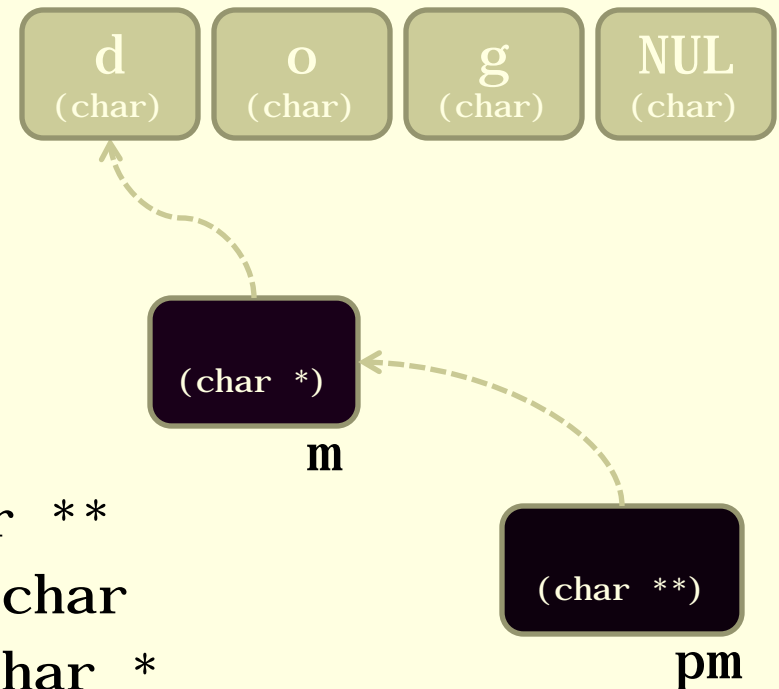
So,   **pm** needs a value of type **char ** * * **

■ Can we give it ***m**? No – type is **char**

■ Can we give it **m**? No – type is **char ***

■ **&m** gives it the right value – the *address* of a **char *** value
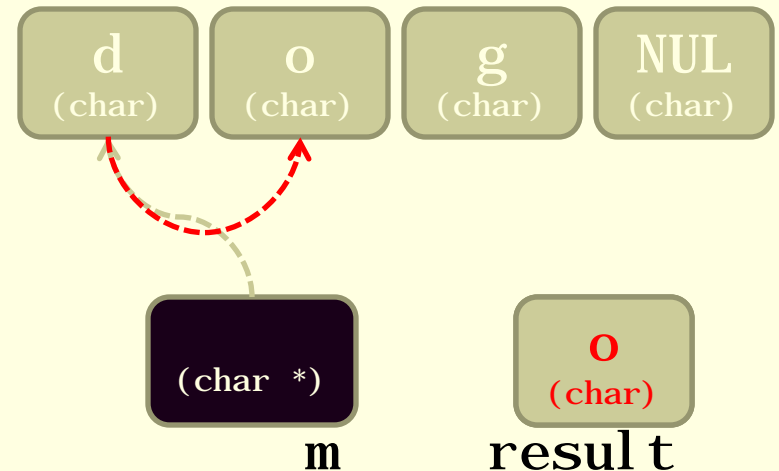
   ■ Just accept it!

Pointer arithmetic

▶ C allows pointer values to be incremented by integer values

```
char   *m = "dog";

char result = *(m + 1);
```


Analyze this code:

m gives an address of a char

(m + 1) gives the char one byte higher

*(m + 1) instructs us to take the contents of that address

result gets the value ′o′

Pointer arithmetic

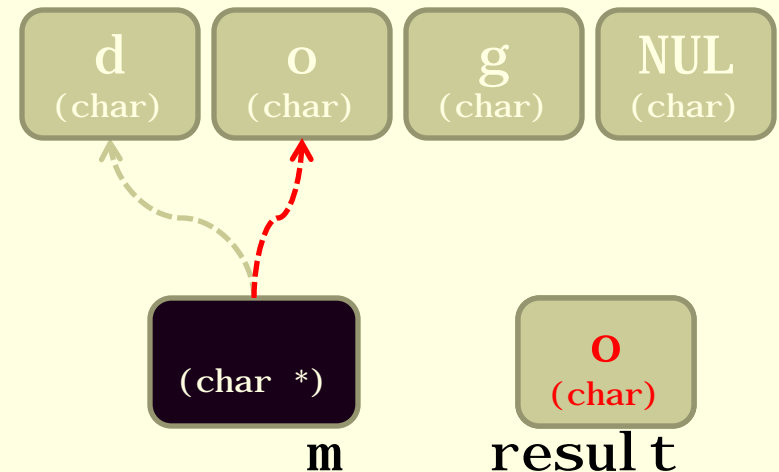■ A slightly more complex example:

```
char   *m = "dog";

char result = *++m;
```

d (char)	o (char)	g (char)	NUL (char)

(char *)

m

o (char)

result

Analyze this code:

m gives an address of a char

++m changes m, to the address one byte higher,
    and returns the new address

*++m instructs us to take the contents of that location

result gets the value ′o′

Review of pointers

- Again:

- What is a Pointer?

■An Address!

Pointer arithmetic

▸ How about multibyte values?

   ▸ **Q:** Each `char` value occupies exactly one byte, so obviously incrementing the pointer by one takes you to a new `char` value... But what about types like `int` that span more than one byte?

   ▸ **A:** C "does the right thing": increments the pointer by the size of one `int` value

17 (int)		**42** (int)

```
int   a[2] = {17, 42};
int *m = a;
int result = *++m;
```
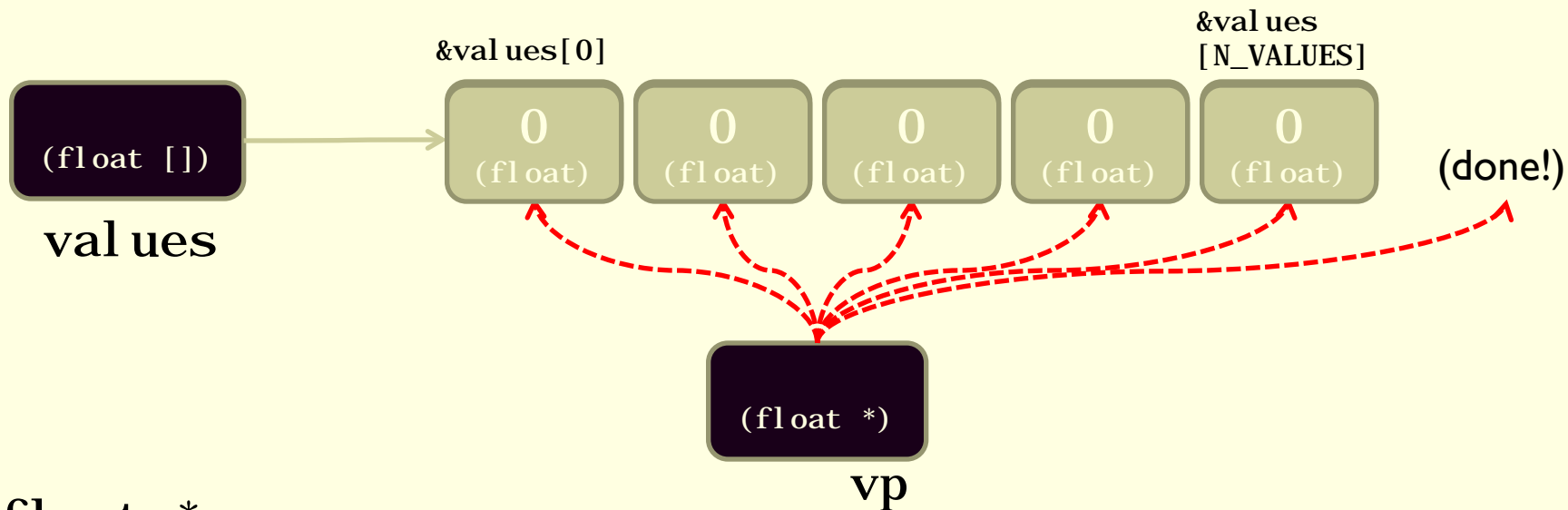
(int *)

m

42
(int)

result

Example: initializing an array

```
#define N_VALUES 5
float values[N_VALUES];
```
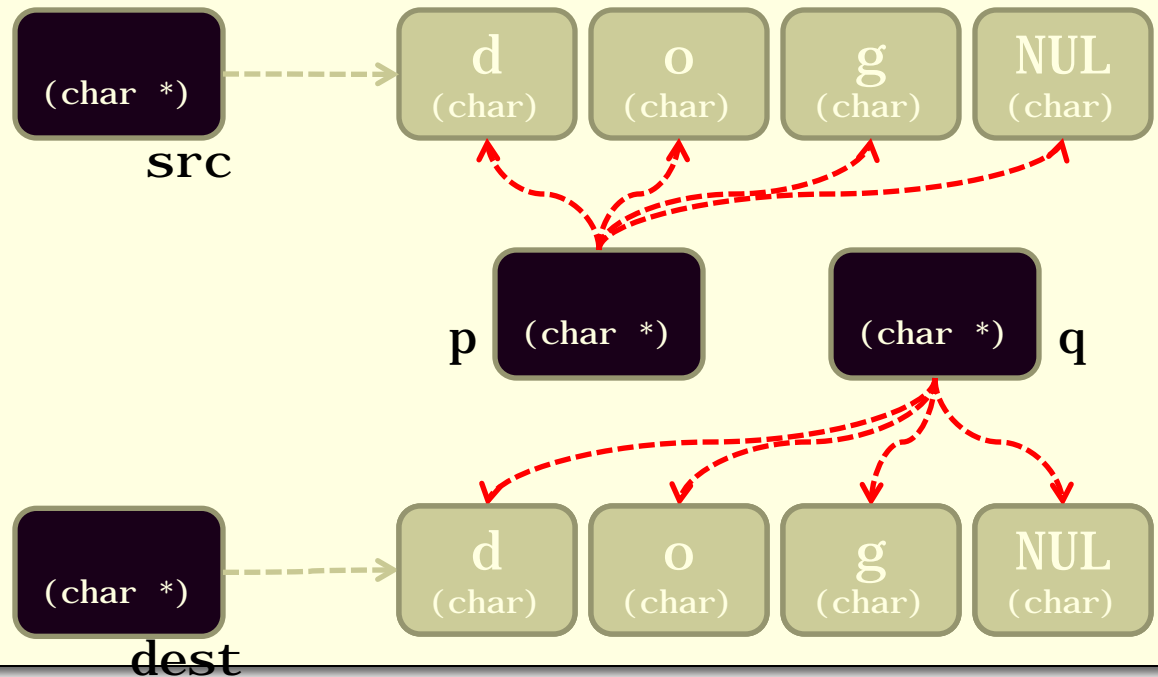


```
float *vp;
for ( vp = &values[0]; vp < &values[N_VALUES]; )
    *vp++ = 0;
```

Example: `strcpy` "string copy"

```
char *strcpy(char *dest, const char *src) {
    const char *p;
    char *q;
    for(p = src, q = dest; *p != '\0'; p++, q++)
        *q = *p;
    *q = '\0';
    return dest;
}
```

(char *) ----→ d (char) | o (char) | g (char) | NUL (char)

src

p (char *)   (char *) q

(char *) ----→ d (char) | o (char) | g (char) | NUL (char)

dest

Review of pointers

■ One final time:

■ What is a Pointer?

■An Address!

C-Programming Review

ARRAYS

Review of arrays

- There are no array variables in C – only array *names*
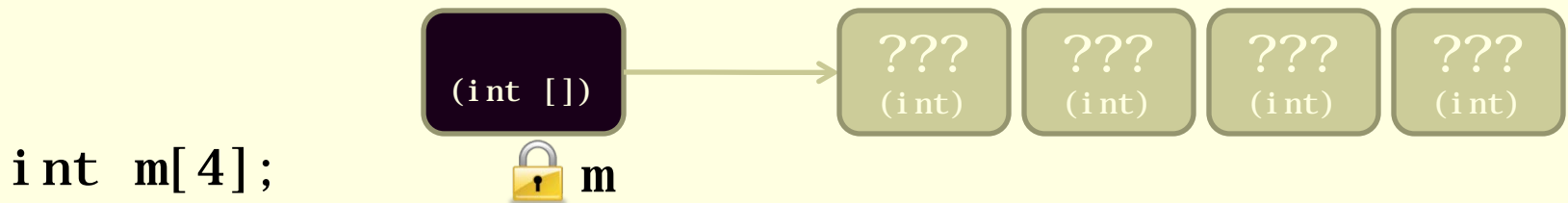 - Each name refers to a **constant <u>pointer</u>**

Review of arrays

Review of arrays

- There are no array variables in C – only array *names*
    - Each name refers to a **constant <u>pointer</u>**
    - Space for array elements is allocated at declaration time
- Can't change where the array name refers to…
    - but you can change the array elements,

        via **<u>pointer</u> arithmetic**

```
(int [])
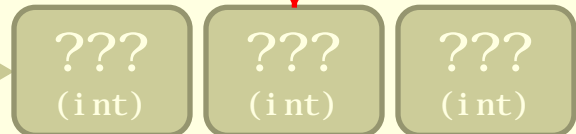```
```
???   ???   ???   ???
(int)   (int)   (int)   (int)
```

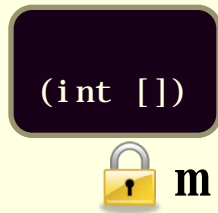`int m[4];`   🔒 **m**

# Subscripts and pointer arithmetic

- `array[subscript]` equivalent to `*(array + (subscript))`

- Strange but true: Given earlier declaration of `m`, we all know that the expression m[2] is legal

- But also, the expression `2[m]` is legal!
  - Not only that:  it's equivalent to
  - `*(2+m)`
  - `*(m+2)`
  - `m[2]`
  - These all point to the same thing!

# Array names and pointer variables, playing together
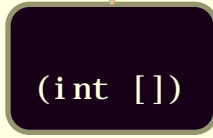
```
int m[3];
```
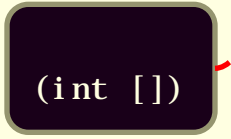
subscript OK with pointer variable

`(int [])`

🔒 m

??? (int)  ??? (int)  ??? (int)

```
int *mid = m + 1;
```

`(int [])`

mid

```
int *right = mid[1];
```

`(int [])`
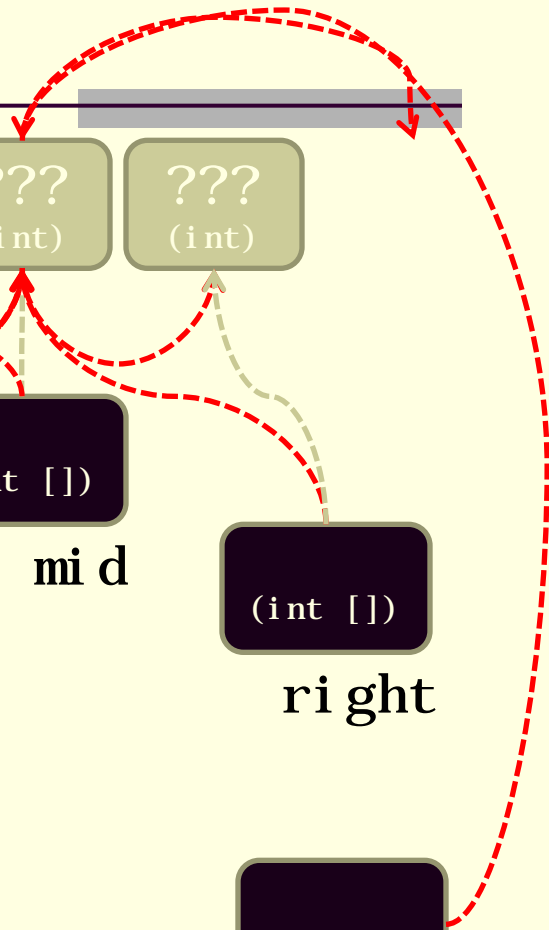
right

```
int *left = mid[-1];
```

`(int [])`

left

```
int *beyond = mid[2];
```

`(int [])`

beyond

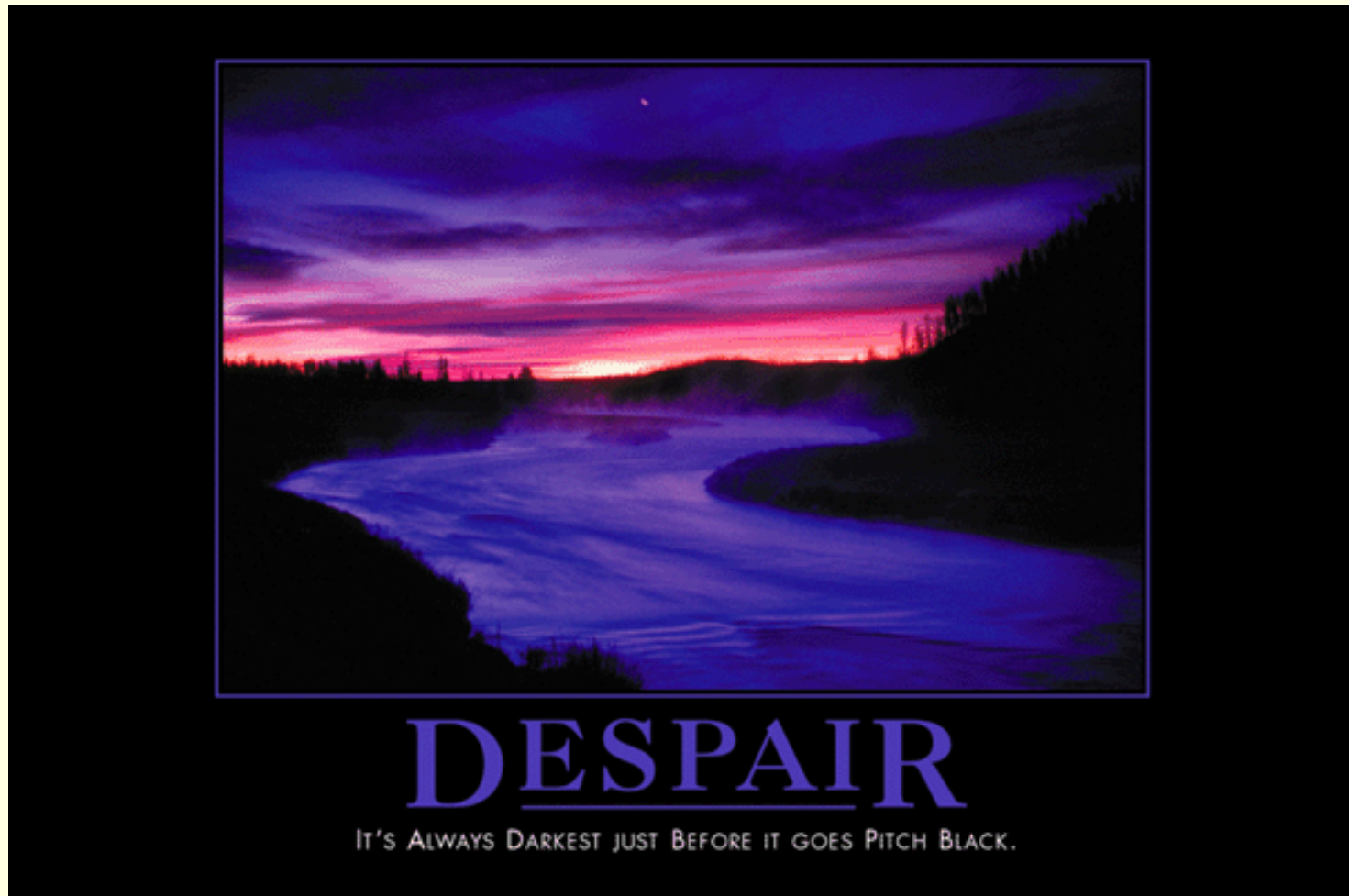compiler may not catch this – runtime environment certainly won't

A Gift for all of you

Demotivator Time

C-Programming Review Pointers & Arrays

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I