COS 217: Introduction to Programming Systems

Pointers, Arrays, and Strings

PRINCETON UNIVERSITY

POINTERS

Pointers in C

So… what's a pointer?

- A pointer is a variable
- Its value is the *location* of another variable
- "Dereference" or "follow" the pointer to read/write the value at that location

Why is *that* a good idea?

- Copying large data structures is inefficient; copying pointers is fast
- x=y is a one-time copy: if y changes, x doesn't "update"
- Parameters to functions are *copied*; but handy to be able to modify value
- Often need a handle to access dynamically allocated memory

Straight to the Point

Pointer types are target dependent

- Example: “int *p;” – declares p to be a pointer to an int
- We’ll see “generic” pointers later

Values are memory addresses

- … so size is architecture-dependent – 8 bytes on ARMv8
- NULL macro in stddef.h for special pointer guaranteed not to point to any variable

Pointer-specific operators

- Address-of operator (&) – creates a pointer
- Dereference operator (*) – follows a pointer

Other pointer operators

- Assignment operator: =
- Relational operators: ==, !=, >, <=, etc.
- Arithmetic operators: +, –, ++, –=, !, etc.

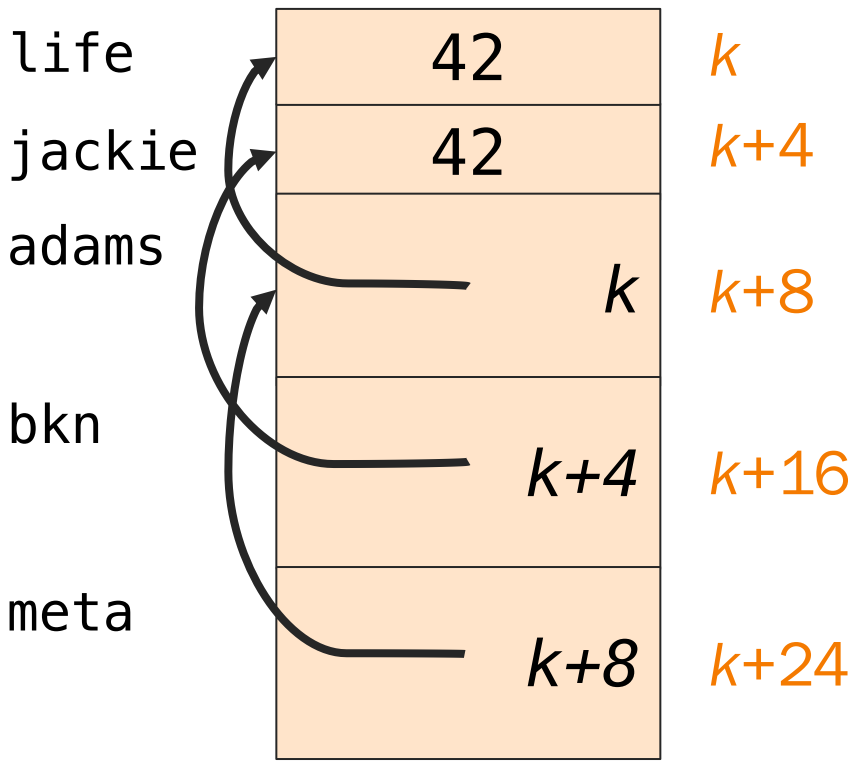
To Illustrate the Point…

```
int life = 42;
int jackie = 42;
int *adams = &life;
int *bkn = &jackie;
int **meta = &adams;
printf("%d %d\n",
             adams == bkn,
            *adams == *bkn);
```

```
0 1
```

```
printf(”%d %d %d %d %d\n",
             meta == &adams,
             meta == &bkn,
            *meta == adams,
            *meta == bkn,
           **meta == *bkn);
```

```
1 0 1 0 1
```



What Points to What?

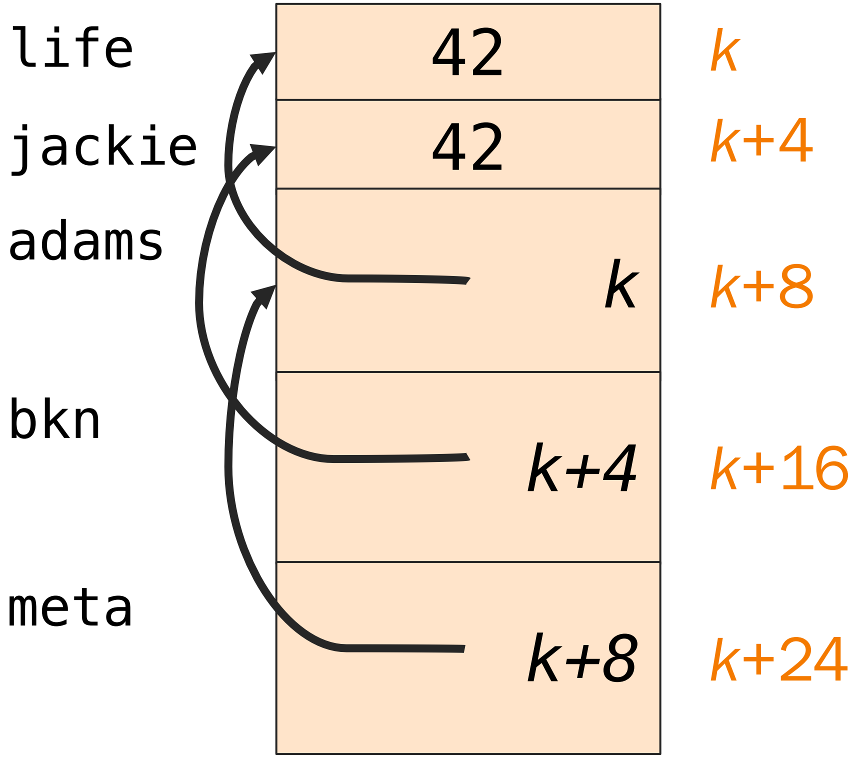
```
adams = bkn;


printf("%d %d\n",
             adams == bkn,
            *adams == *bkn);
```

A: 0 0

B: 0 1

C: 1 0

D: 1 1



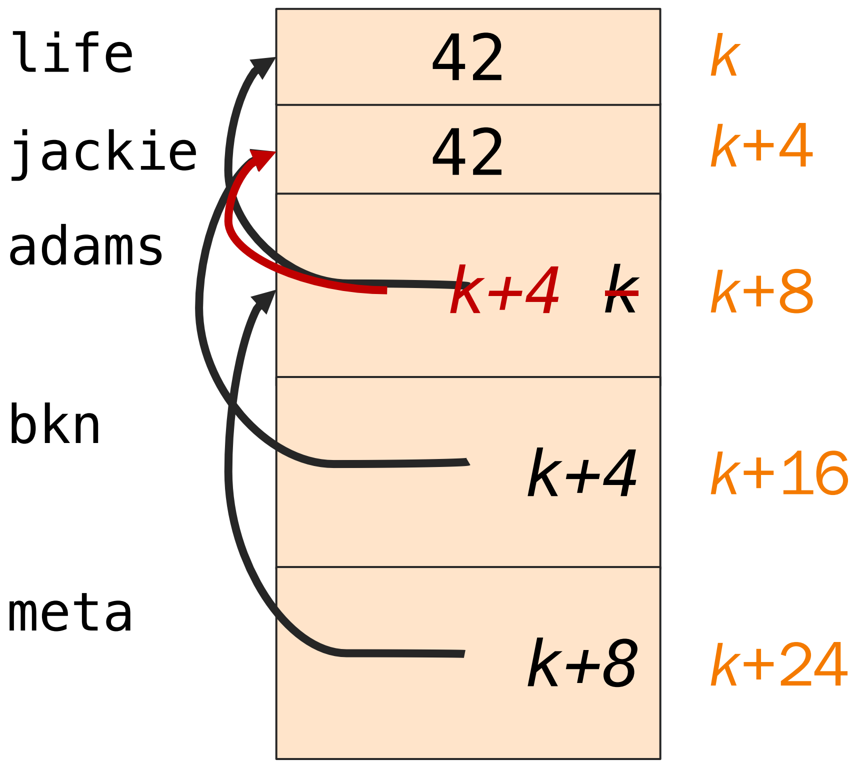
What Points to What?

```
adams = bkn;


printf("%d %d\n",
             adams == bkn,
            *adams == *bkn);
```

1 1

```
printf(”%d %d %d %d %d\n",
             meta == &adams,
             meta == &bkn,
            *meta == adams,
            *meta == bkn,
           **meta == *bkn);
```

1 0 1 1 1



life 42 k

jackie 42 k+4

adams k+4 ~~k~~ k+8

bkn k+4 k+16

meta k+8 k+24

Pointer Declaration Gotcha

Pointer declarations can be written as follows: `int* p;`

This is equivalent to: `int *p;`

but the former seemingly emphasizes that the *type* of p is (int *).

Even though this syntax seems more natural, and you are welcome to use it, it isn’t how the designers of C thought about pointer declarations.

So beware! This declaration: `int* p1, p2;`

really means: `int *p1;  int p2;`

To declare both p1 and p2 as pointers, need: `int* p1;  int* p2;`

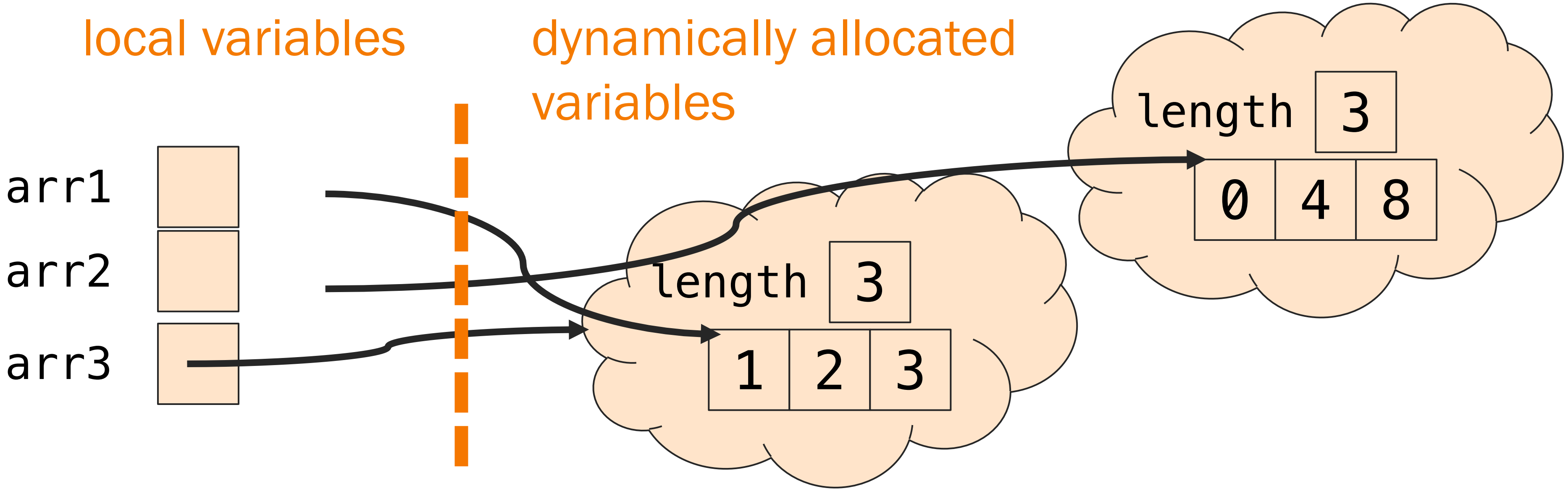
Or, the following works: `int *p1, *p2;`

ARRAYS

Refresher: Java Arrays

- Always dynamically allocated
  - Even when the values are known at compile time (e.g. initializer lists)
- Access via a reference variable

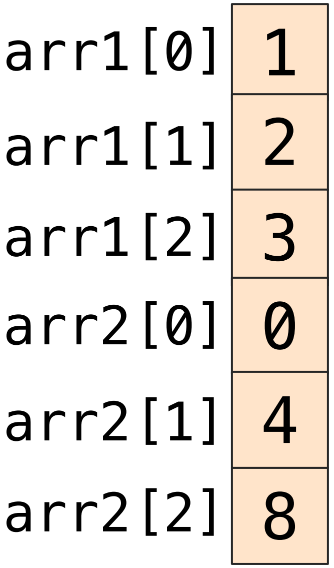
```
public static void arrays() {
  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];
  for(int c = 0;
          c < arr2.length; c++)
        arr2[c] = 4*c;
  int[] arr3 = arr1;

}
```



C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
      sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```



arr1[0]	1
arr1[1]	2
arr1[2]	3
arr2[0]	0
arr2[1]	4
arr2[2]	8

C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

arr1[0]	1
arr1[1]	2
arr1[2]	3
arr2[0]	0
arr2[1]	4
arr2[2]	8

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
       sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```

C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

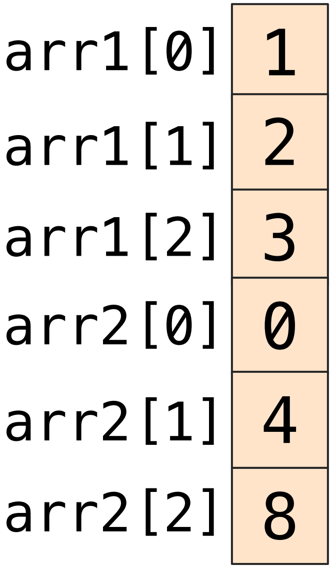
```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
      sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```

arr1[0]	1
arr1[1]	2
arr1[2]	3
arr2[0]	0
arr2[1]	4
arr2[2]	8

C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
        sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```

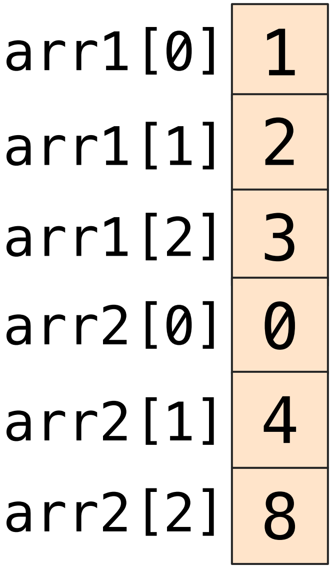


arr1[0]	1
arr1[1]	2
arr1[2]	3
arr2[0]	0
arr2[1]	4
arr2[2]	8

C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won't see this until Lecture 8

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
       sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```



arr1[0]	1
arr1[1]	2
arr1[2]	3
arr2[0]	0
arr2[1]	4
arr2[2]	8

Pointer/Array Interplay

- Array name alone can be used as a pointer: `arr` vs. `&arr[0]`

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
       sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
```

```
  int *arr3 = arr1;
        /* or */
  int *arr3 = &arr1[0];
```

Pointer/Array Interplay

- Array name alone can be used as a pointer: `arr` vs. `&arr[0]`
- Subscript notation can be used with pointers

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  int arr2len =
      sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
       arr2[c] = 4*c;
  ~~int[] arr3 = arr1;~~
}
              ↓

  int *arr3 = arr1;
  int i = arr3[1];
```

Pointer Arithmetic

Array indexing is actually a pointer operation!

`arr[k]` is syntactic sugar for `*(arr + k)`

Implies that pointer arithmetic is on elements, not bytes:

`ptr ± k` is implicitly
`ptr ± (k * sizeof(*ptr))` bytes

Subtracting two pointers gives you a count of elements, not bytes:

```
(ptr + k) – ptr == k
```

Arrays with Functions

Passing an array to a function

- Arrays "decay" to pointers (the function parameter gets the address of the array)
- Array length in signature is ignored
- `sizeof` "doesn't work"

```
/* equivalent function signatures */
size_t count(int numbers[]);
size_t count(int *numbers);
size_t count(int numbers[5]);
{

    /* always returns 8 */
    return sizeof(numbers);
}
```

Returning an array from a function

- C doesn't permit functions to have arrays for return types
- Can return a pointer instead
- Be careful not to return an address of a local variable (since it will be deallocated!)

~~int[] getArr();~~

```
int *getArr();
```

STRINGS

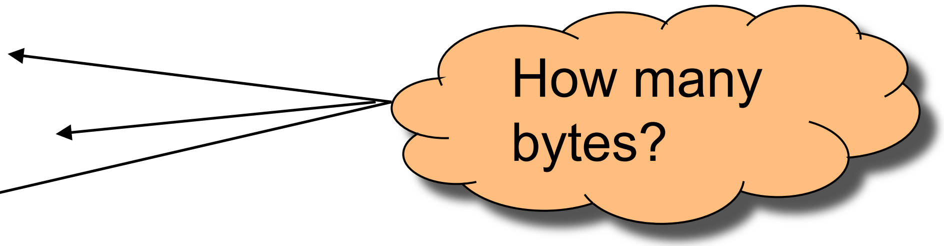
Strings and String Literals in C

A string in C is a sequence of contiguous chars

- Terminated with null char ('\0') – not to be confused with the NULL pointer
- Double-quote syntax (e.g., "hello") to represent a string literal
- String literals can be used as special-case initializer lists
- No other language features for handling strings
  - Delegate string handling to standard library functions

Examples

- 'a' is a char literal
- "abcd" is a string literal
- "a" is a string literal



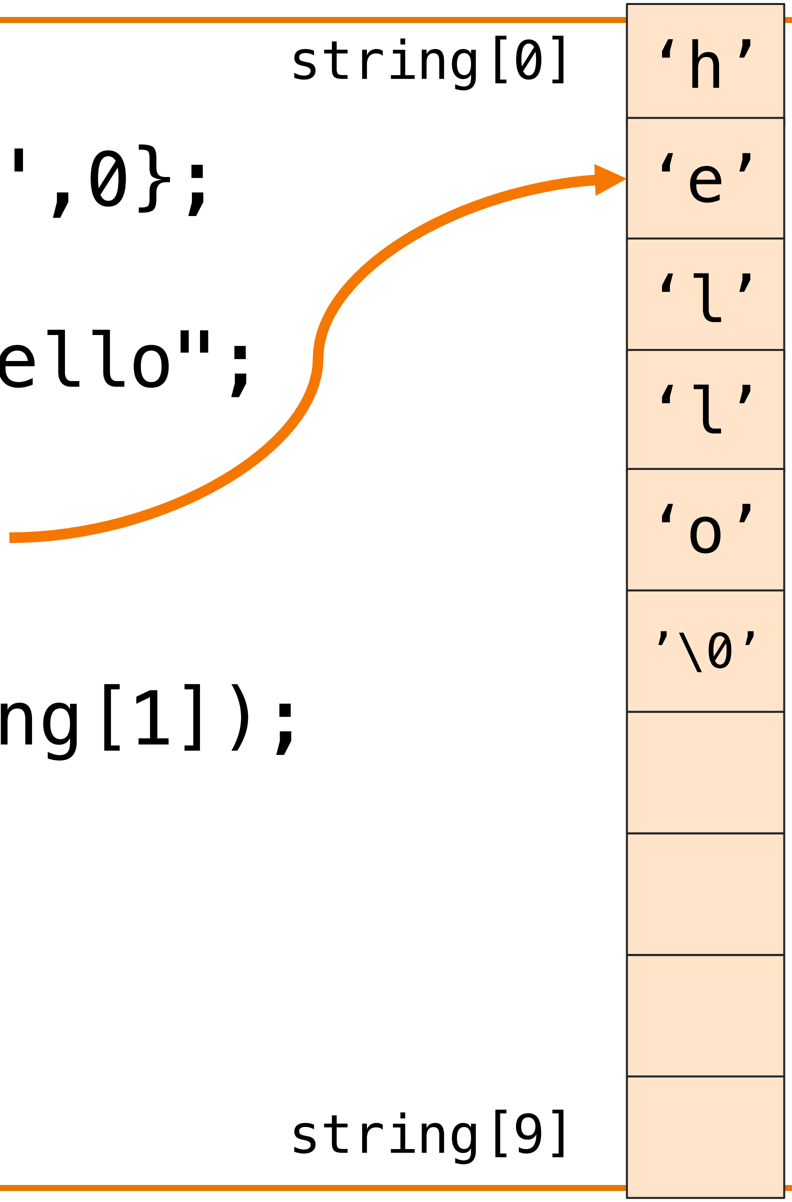
How many bytes?

Lemon Gelatin Dessert

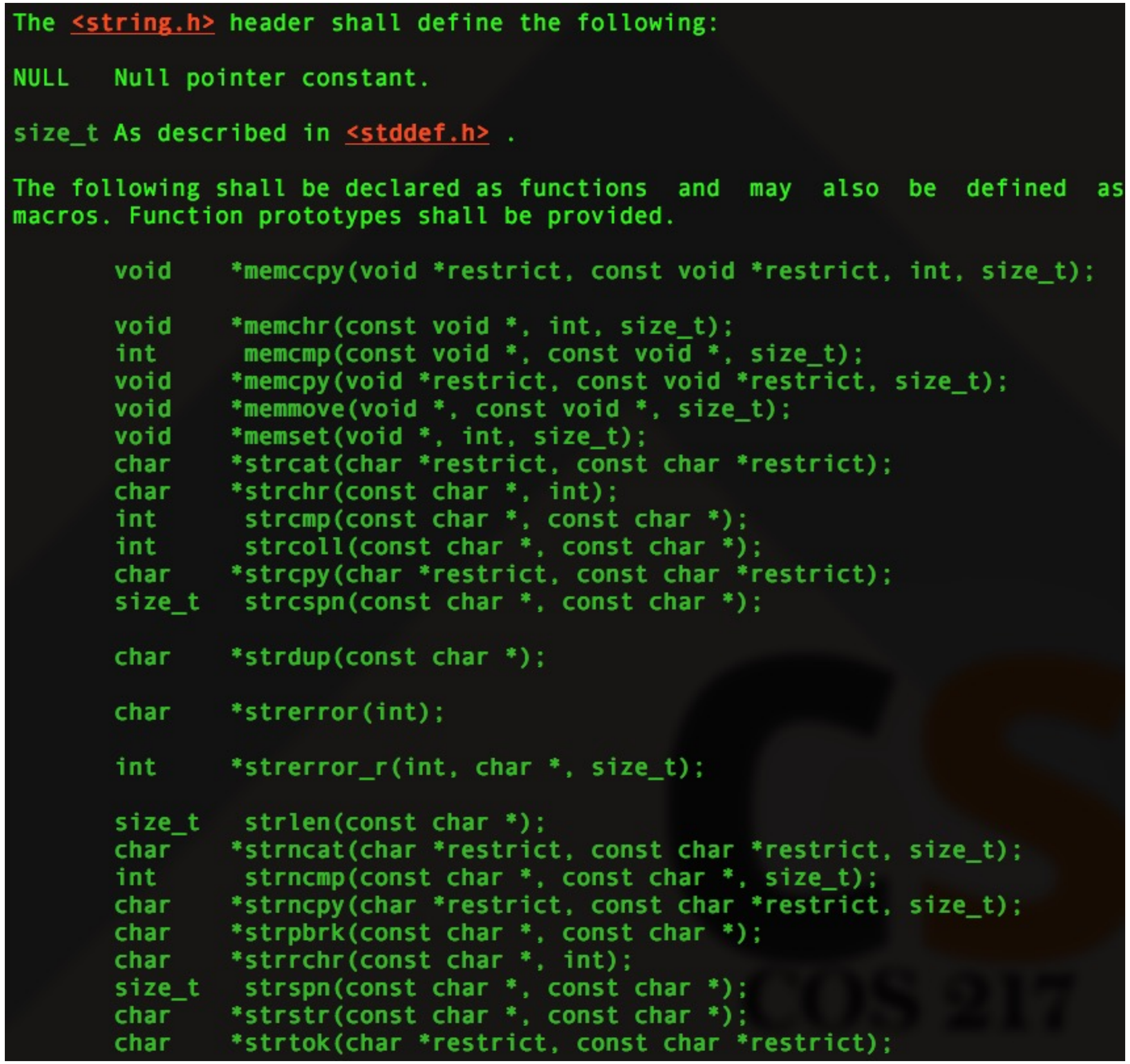
```
char string[10] =
 {'H','e','l','l','o',0};
```

(or, equivalently)

```
char string[10] = "Hello";

char *pc = string+1;

printf(”Y%s ", &string[1]);
printf("J%s!", pc);
```



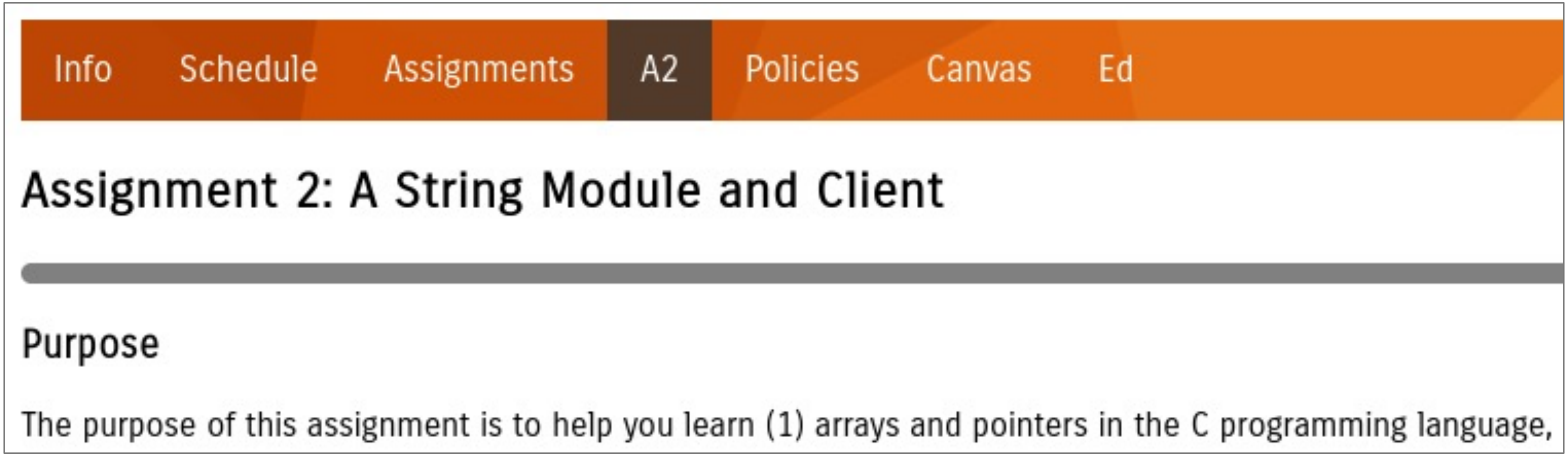
Standard String Library



```
The <string.h> header shall define the following:

NULL   Null pointer constant.

size_t As described in <stddef.h> .

The following shall be declared as functions  and  may  also  be  defined  as
macros. Function prototypes shall be provided.

       void    *memccpy(void *restrict, const void *restrict, int, size_t);

       void    *memchr(const void *, int, size_t);
       int      memcmp(const void *, const void *, size_t);
       void    *memcpy(void *restrict, const void *restrict, size_t);
       void    *memmove(void *, const void *, size_t);
       void    *memset(void *, int, size_t);
       char    *strcat(char *restrict, const char *restrict);
       char    *strchr(const char *, int);
       int      strcmp(const char *, const char *);
       int      strcoll(const char *, const char *);
       char    *strcpy(char *restrict, const char *restrict);
       size_t   strcspn(const char *, const char *);

       char    *strdup(const char *);

       char    *strerror(int);

       int     *strerror_r(int, char *, size_t);

       size_t   strlen(const char *);
       char    *strncat(char *restrict, const char *restrict, size_t);
       int      strncmp(const char *, const char *, size_t);
       char    *strncpy(char *restrict, const char *restrict, size_t);
       char    *strpbrk(const char *, const char *);
       char    *strrchr(const char *, int);
       size_t   strspn(const char *, const char *);
       char    *strstr(const char *, const char *);
       char    *strtok(char *restrict, const char *restrict);
```

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
enum { LENGTH = 14 };
int main() {
  char h[] = "Hello, ";
  char w[] = "world!";
  char msg[LENGTH];
  char ∗found;
  if(sizeof(msg) <= strlen(h) + strlen(w))
    return EXIT_FAILURE;
  strcpy(msg, h);
  strcat(msg, w);
  if(strcmp(msg),
           "Hello, world!"))
    return EXIT_FAILURE;
  found = strstr(msg, ", ");
  if(found – msg != 5)
    return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

DIY (x2) – Available Later This Week



Assignment 2: A String Module and Client

Purpose

The purpose of this assignment is to help you learn (1) arrays and pointers in the C programming language,