COS 217: Introduction to Programming Systems

Indirection

Pointers, Arrays, and Strings

PRINCETON UNIVERSITY

Indirection

- **Referencing an object through an intermediary, such as a pointer, handle, or a layer of abstraction, rather than directly**
  - Provides ***decoupling***
  - Which helps manage complexity and provide flexibility in programming and system design
  - Critical in many areas of CS: databases, Internet (e.g. DNS), virtual machines, design patterns, …

- "All problems in computer science can be solved by another level of indirection”

- But can get complicated if not managed well, and can reduce performance

- Examples?

Indirection Examples

- URLs
- Search indexes: can change index, or indexing method, without changing data
- DNS: can change name of web site, or relocate it, without affecting links or underlying systems
- Virtual machines: write programs for VM, run them on any hardware

Indirection with URLs

- **Different names for same object**
  - cocacola.com, coca-cola.com, ~~coke.com~~
  - Names that point to different objects at different times: leadingsoftdrink.com
- Access with different protocols, e.g. security
  - http://www.coca-cola.com, https://www.coca-cola.com
- Access with different parameters
  - User context, analytics parameters
- Access from different subdomains

Pointers are a key example of indirection in programming

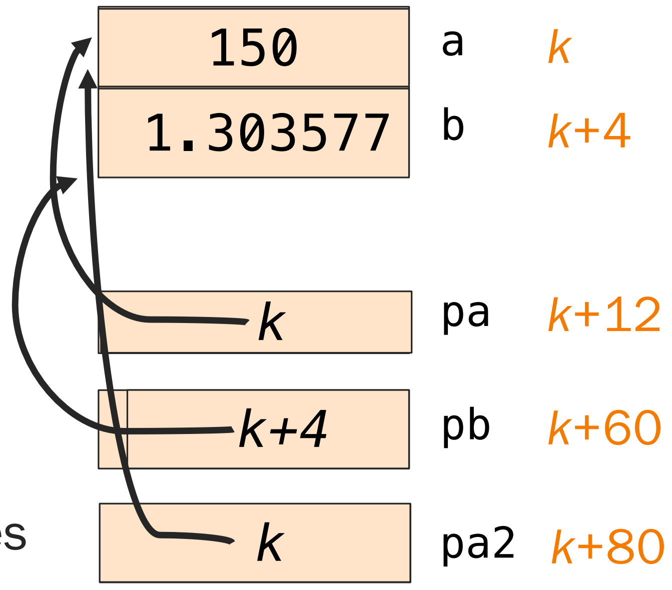
Agenda

- Pointers
- Arrays
- Strings

Pointers in C

So… what's a pointer?

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



Why is *that* a good idea?

- Copying pointers is much faster than large structures
- 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

Pointers in C

Pointer types are target dependent

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

Pointer values are memory *addresses*

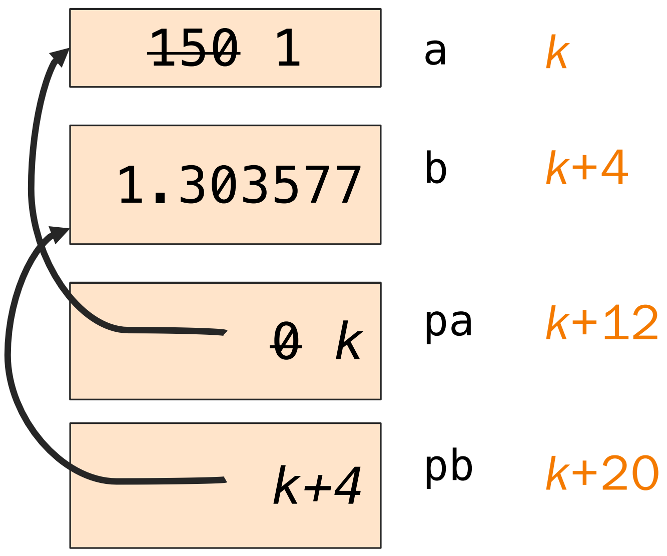
- … so size is architecture-dependent – 8 bytes on ARMv8
- NULL is macro in stddef.h for special pointer guaranteed not to point to any variable

Pointer-specific operators

- Address-of operator (&a) – a pointer to variable “a”
- Dereference operator (*a) – value that ptr “a” points to

Other pointer operators

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



```
 int a = 150;
 double b = 1.303577;
 int *pa = NULL;
 double *pb = &b;
 pa = &a;
*pa = (int) *pb;
```

Pointers in C

```
int a = 42;

int b = 42;

int *pa = &a;

int *pb = &b;

int **pc = &pa;

printf("%d %d\n",
              pa ==  pb,
             *pa == *pb);
```

0 1

```
printf(”%d %d %d %d %d\n",
              pc == &pa,
              pc == &pb,
             *pc ==  pa,
             *pc ==  pb,
            **pc == *pb);
```

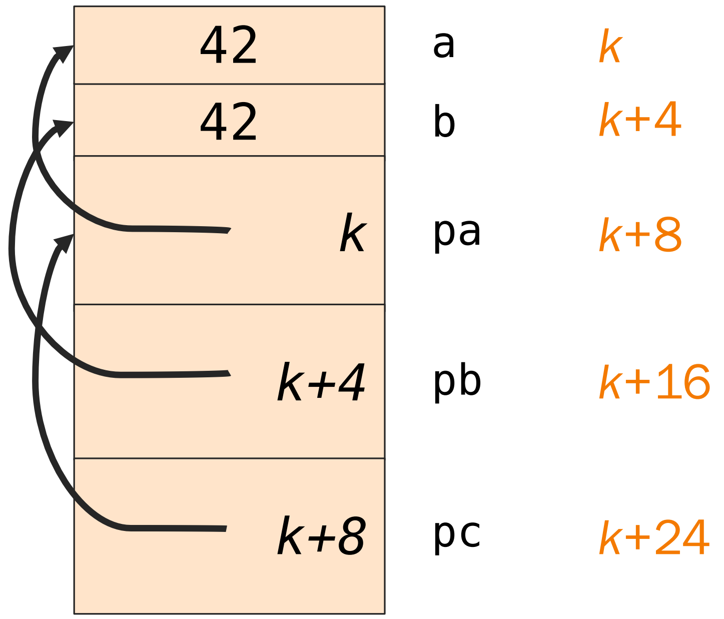
1 0 1 0 1

<- same as *pa == *pb

What if this were:

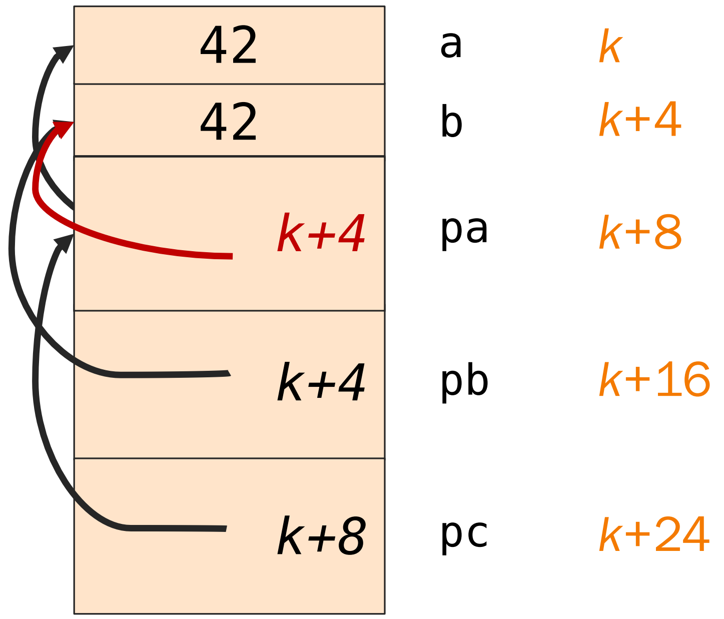
```
int *pa;
*pa = &a;
```

Is it the same?



What Points to Whom, Where?

```
pa = pb;

printf("%d %d\n",
              pa ==  pb,
             *pa == *pb);
```



A: 0 0
B: 0 1
C: 1 0
D: 1 1

Pointer Declaration Gotcha

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

This is equivalent to: `int *pi;`

but the former seemingly emphasizes that the *type* of pi is ("int pointer")

Even though the first syntax may seem more natural, and you are welcome to use it, the * is attached to the variable name (pi), not to the type.

Beware!!!!! This declaration: `int* p1, p2;`

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

To declare both p1 and p2 as pointers, i.e.: `int* p1;  int* p2;`

in one statement, must "star" both variables: `int *p1, *p2;`

Agenda

- Pointers
- Arrays
- Strings

Non Sequitur

Options for lecture the Monday of Thanksgiving week:

A. Re-visit some topics with material we didn't get to on the lecture slides

B. Walk-through of some trickier iClicker questions and some past exam questions

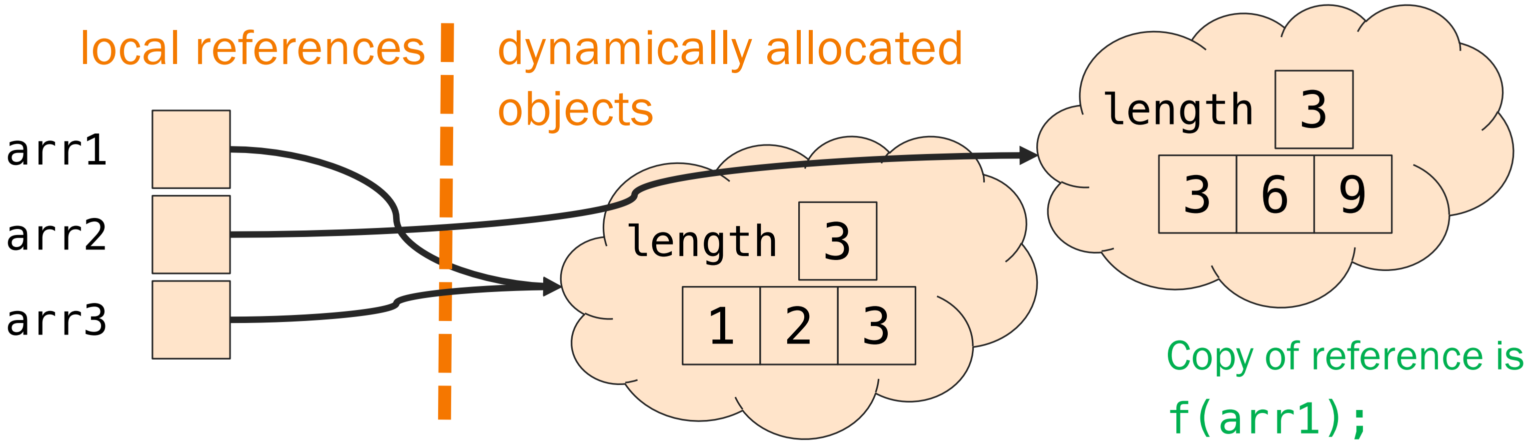
C. Cover some OS material (processes, signals, pipes) that used to be in COS217

D. Cancel lecture (take the week off from 217: no other 217 activities that week)

Refresher: Java Arrays

- Always dynamically allocated
  - Even when the values are known at compile time (e.g., initializer lists)
  - Objects on the heap
- Access via reference variable (handle)
  - Dereference, assign/copy. Can't see value

```
public static void arrays() {
  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];
  int[] arr3 = arr1;

  for (int c = 0;
        c < arr2.length; c++)
      arr2[c] = 3 * arr1[c];
}
```



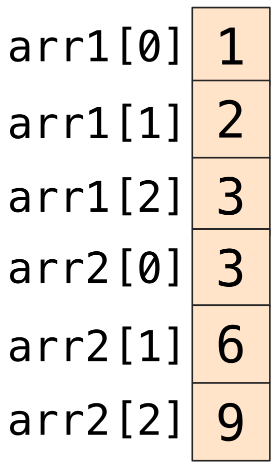
Copy of reference is also done in:

`f(arr1);`

C Arrays

- Can be *statically allocated* e.g., as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We will see this in Lecture 8

```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];
  ~~int[] arr3 = arr1;~~

  int arr2len =
       sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
        arr2[c] = 3 * arr1[c];
  }
```



Java and C Arrays

```
public static void arrays() {

  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];

  for(int c = 0;
        c < arr2.length; c++)
      arr2[c] = 3 * arr1[c];


  int[] arr3 = arr1;
  }
```

```
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] = 3 * arr1[c];
```

~~int[] arr3 = arr1;~~

```
}
```

Can we declare this way?

`int c, arr1[] = {1, 2, 3};`

Java and C Arrays

```
public static void arrays() {

  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];

  for(int c = 0;
        c < arr2.length; c++)
      arr2[c] = 3 * arr1[c];


  int[] arr3 = arr1;
  }
```

```
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] = 3 * arr1[c];

  ~~int[] arr3 = arr1;~~
}
```

Unlike in Java, local variables are not automatically initialized to 0 in C.
So can contain junk unless initialized.

A Programming Note …

Yes, there’s a midterm.

Wednesday, October 8. (2 weeks from today)

In class. So, 50 minutes. We start at the start. **Please don’t be late**

On paper. Closed book. 1 one-sided study sheet allowed.

Covers through Thursday 10/2. Exam info page available now on course web site. **Please read it carefully**

Java and C Arrays

```
public static void arrays() {

  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];

  for(int c = 0;
        c < arr2.length; c++)
      arr2[c] = 3 * arr1[c];


  int[] arr3 = arr1;
  }
```

```
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] = 3 * arr1[c];

  ~~int[] arr3 = arr1;~~
}
```

C arrays don[t have a length field, since they're not objects, just addresses
This use of sizeof to get length only works for fixed-size arrays, within the scope where declared
Note that in C, unlike in Java, can't declare a variable inside the loop control

Java and C Arrays

```
public static void arrays() {

  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];

  for(int c = 0;
        c < arr2.length; c++)
      arr2[c] = 3 * arr1[c];


  int[] arr3 = arr1;
  }
```

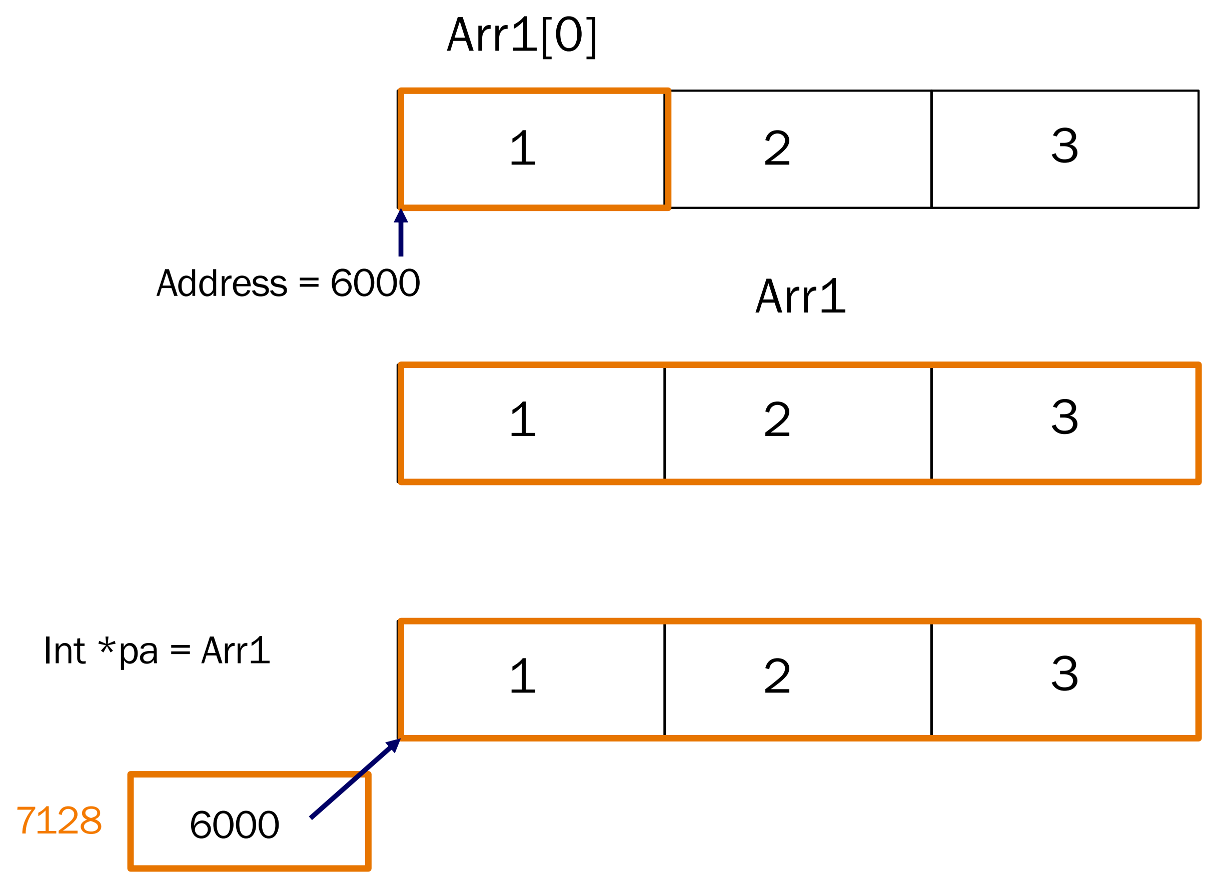
```
void arrays() {
  int c;
  int arr1[] = {1, 2, 3};
  int arr2[3];

  int arr2len =
      sizeof(arr2)/sizeof(int);
  for (c = 0; c < arr2len; c++)
        arr2[c] = 3 * arr1[c];

  ~~int[] arr3 = arr1;~~
}
```

Array names can't be assigned new values. This neither copies pointers nor the entire object. It's just wrong. But could use pointers: `pa = arr1;`

Pointers and Arrays: An Array Name is Not a Pointer

- Arrays in C are not pointers. They're data types with variable sizes
  - sizeof(arr1) is 3*sizeof(int); it is not 1 * sizeeof(int *)
  - An array is just a composite of its elements. There isn't another object
- An array name is the address of this array. first byte of the array
  - So, like a pointer in that sense. `arr1` is the same as `&arr1[0]`
- **Unlike pointer, there is no variable in memory that stores the address**
  - Can't assign a new value to it: can't say `arr1 = arr2;` or `arr1 = pa;`
  - `&arr1` is addr of the composite, so same value as `arr1` or `&arr1[0]`
  - Can't take the address of the address of an array name (or of a pointer)
    - ~~`pa = &(&arr1); pa = &(&pb);`~~

Arrays in C



Print &Arr1[0] = 6000
Print Arr1[0] = 1

Print &Arr1 = 6000
Print Arr1 = 6000

Print &pa = 7128
Print pa = 6000
Print *pa = 1

Pointers and Arrays: An Array Name is Not a Pointer

char array[10];

sizeof(array) == 10	the size of the array
sizeof(*array) == 1	the size of the pointed to first element (as by def)
sizeof(&array) == 8	size of a pointer to the array.
sizeof(*&array) == 10	size of the pointed to object (the array)
sizeof(array[0]) == 1	size of the first element.
sizeof(&array[0]) == 8	size of a pointer to the first element (as above)
sizeof(*&array[0]) == 1	size of the pointed to first element.

But an Array Name is an Address

- Can assign array name to a pointer: `p = arr1; p = &arr1[0];`
- Can use array names in pointer arithmetic

```
int arr1[] = {1, 2, 3};
int *p = arr1 + 10;
```

  - *Note: Arithmetic on pointers is in units of sizeof(data type), not bytes:*

    `ptr ± k` is implicitly `ptr ± (k * sizeof(*ptr))` bytes
    `(ptr + k) – ptr == k.`

    - True for all pointer types. Also true for array names.

- If a ptr is assigned an array, we can use it like we use the array name
  - If `p = arr1`, `p[k]` is same as `arr[k]`, `p+10` is same as `arr1+10`, etc.

Array Names are Just (Immutable) Addresses

- If `arr1` is an array of int, `arr1 + k` is the address of the kth element of `arr1`, i.e. the address of `arr1[k]`, which is &`arr1[k]`)
- So, `arr1[k]` is the same as `*(arr1 + k)`
- **The C secret: Array indexing is just syntactic sugar**

  `arr[k]` is actually `*(arr + k)`

Can Use Array Names or Pointers to Access Elements

If `int *p = arr1`, can reach elements of `arr1` as

- arr1[0], arr1[1] ... , arr1[k], ...
- *arr1, *(arr1 + 1), ... *(arr1 + k), ...
- p[0], p[1] ... , p[k], ...
- *p, *(p + 1), ... *(p + k), ...

C programs often use both array names and regular pointers to traverse arrays and dereference array elements

(But remember, array names are not pointers: not variables, no storage)

Arrays with Functions

Passing an array to a function

- Is like passing a pointer (the function parameter gets the address of the array)
- Array length in signature is ignored
- `sizeof` on the array "doesn't work" inside a function

```
/* 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

- Not allowed in C
- Can return a pointer instead
- Be careful not to return an address of a local function variable (as it will be deallocated on return)

~~int[] getArr();~~

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

Agenda

- Pointers
- Arrays
- Strings (precepts)

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

- "abcd" is a string literal
- "a" is a string literal

How many bytes?

Contrast

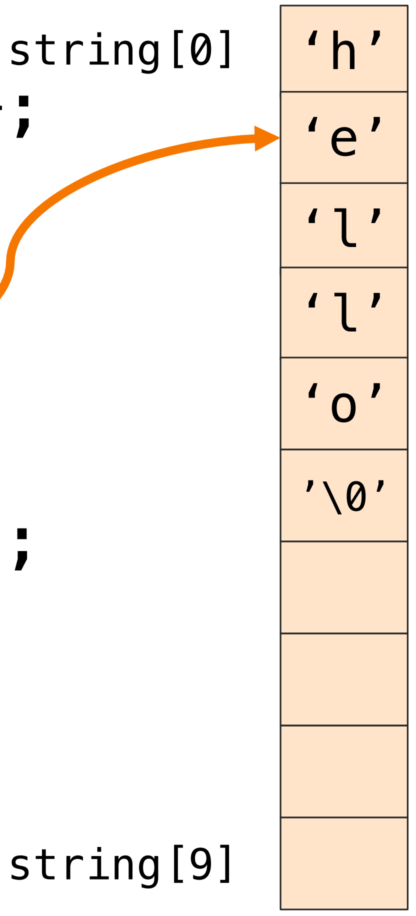
- 'a' is a character literal, not a string literal
  (really an int, as we've discussed)

Pointers for making a Lemon Gelatin Dessert

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

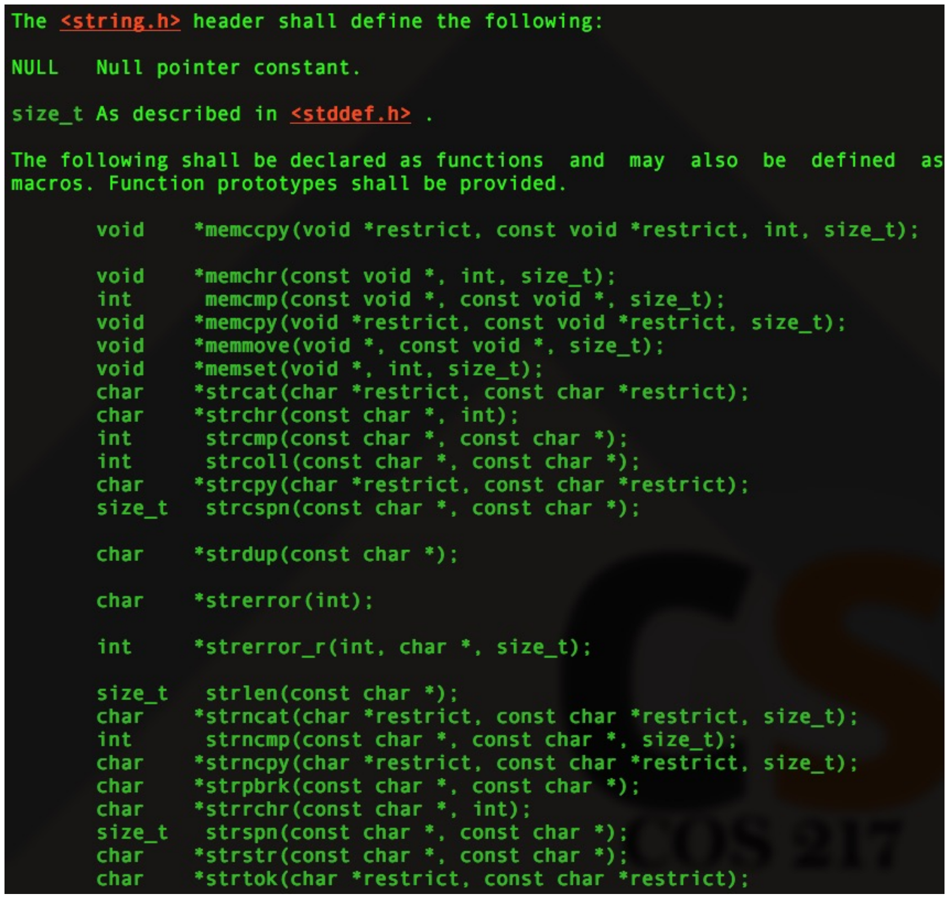
(or, equivalently)*

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

char *pc = string+1;

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



* Unless you mess up counting. See strings.pdf a few precepts from now.

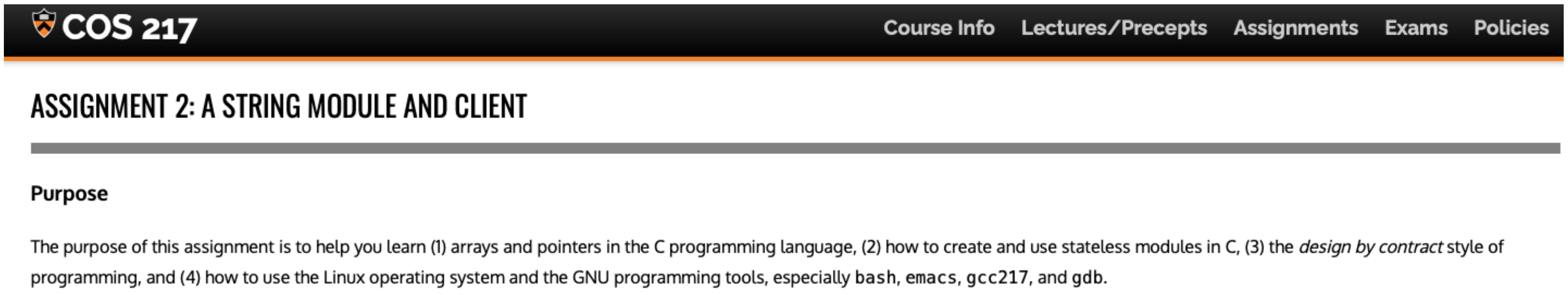
Standard String Library



```
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) – Already Available!



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, (2) how to create and use stateless modules in C, (3) the *design by contract* style of programming, and (4) how to use the Linux operating system and the GNU programming tools, especially `bash`, `emacs`, `gcc217`, and `gdb`.

Sample exam question (Fall 2019)

3b) State concisely what function `q3b()` prints, given a string `pcSrc` as input.

```
void q3b(const char* pcSrc) {
  char* pcCurrent = (char *) pcSrc;
  assert(pcSrc != NULL);

  while (*pcCurrent++ != '\0') ;

  for (pcCurrent--; pcCurrent >= pcSrc; pcCurrent--)
    putchar(*pcCurrent);
}
```

3c) State concisely what function `q3c()` computes and returns, given a string `pcSrc` and a character `c` as input.

```
size_t q3c(const char* pcSrc, char c) {
  size_t local = 0;

  assert(pcSrc != NULL);
  for ( ; *pcSrc != '\0'; pcSrc++)
    local += (*pcSrc == c);
  return local;
}
```

Sample exam question (Fall 2020) 12pts/80

Consider the following function, which copies characters from the source string into the destination string, but unlike in `strcpy`:

* only some characters from the source are copied;
* they're not necessarily consecutive characters from source;
* their order from the source is reversed in the destination.

The destination string must have enough space for the copied characters, but all its allocated space can be assumed to be filled with `'\0'` bytes, such as if it were allocated with `calloc`. If the source string is `NULL` or the empty string, then no change is made to the destination string and the function returns `NULL`.

For example, assuming `buf` has enough space:

`fun(buf, "ab")` should result in `buf: "b"`
`fun(buf, "abc")` should result in `buf: "b"`
`fun(buf, "abcd")` should result in `buf: "cb"`
`fun(buf, "abcde")` should result in `buf: "cb"`
`fun(buf, "abcdefghijklmnopqrstuvwxyz")` should result in `buf: "ngdb"`

```
1 #include <string.h>
2 #include <assert.h>
3
4 char* fun(char* d, char* s) {
5   int i, j;
6   assert(sizeof(d) > sizeof(s));
7   if(strlen(s) == 0 || s == NULL)
8     return NULL;
9   for(i = strlen(s); i /= 2; ) {
10    d[j++] = s[i];
11  }
12  return d;
13 }
```

It has 4 bugs that cause warnings or errors from `gcc217`, runtime crashes, or incorrect contents copied into the destination string.

Identify these bugs by line, with a short description of the problem (e.g. "infinite loop", "accesses array out of bounds", "missing ; after do-while loop") and how it could be fixed (e.g. "start i at 1 not 0", "bound loop with < n not <= n", "add ;", respectively).