COS 217: Introduction to Programming Systems

Crash Course in C (Part 3)

The Design of C Language Features and
Data Types and their Operations and Representations

PRINCETON UNIVERSITY

POINTERS

Pointer Design Decisions

Issue: Why would a variable reference another variable or memory location?

- x=y is a one-time copy: if y changes, x doesn't "update"
- copying large data structures is inefficient
- we need a handle to access dynamically allocated memory

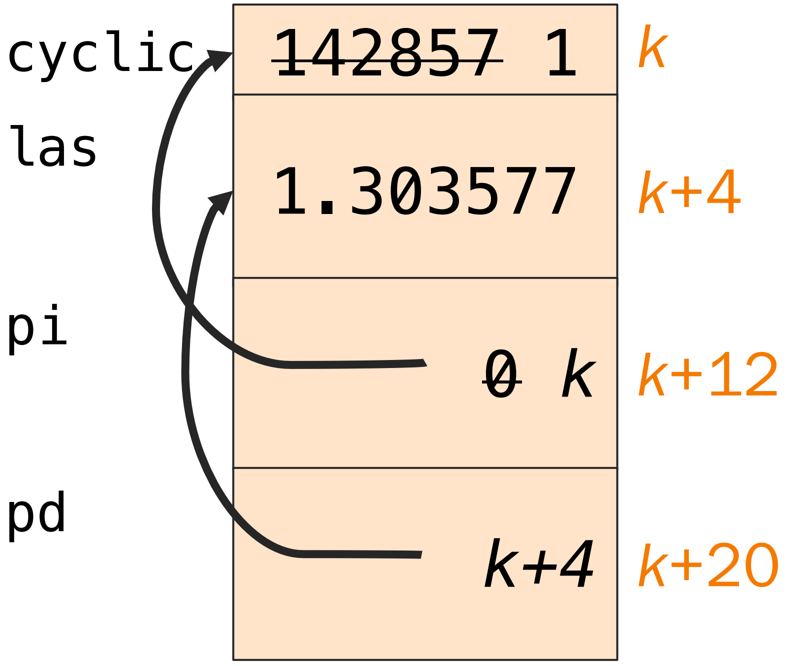
Decision points:

- Typed or generic?
- How to represent a reference?
- What operations are necessary?
  - Create a reference
  - Access the referenced value
  - Reference comparisons?
  - Arithmetic operators for references?

Straight to the Point

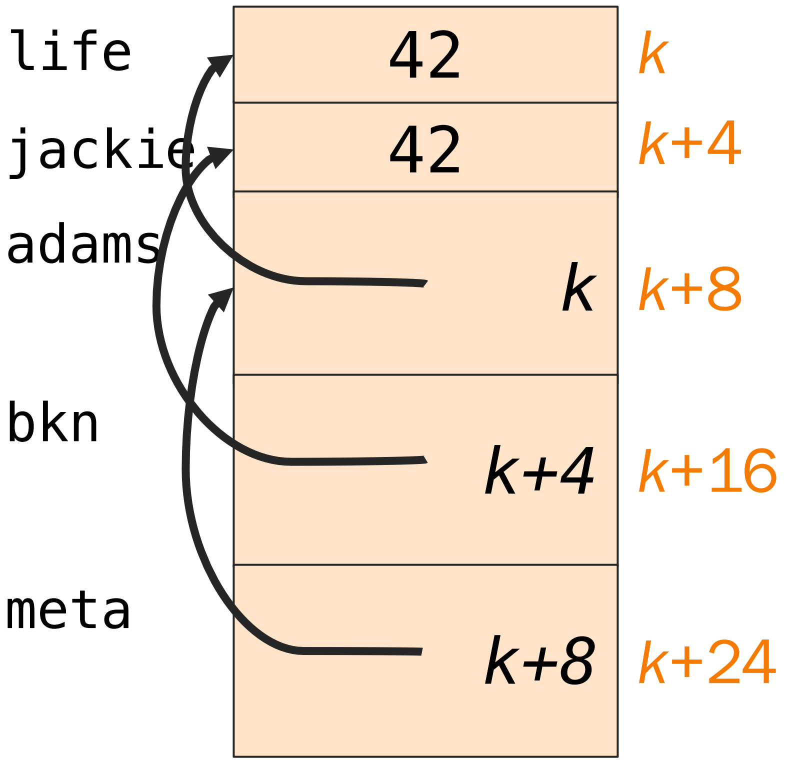
- Types are target-dependent
  - We’ll see “generic” pointers later
- Values are memory addresses
  - so size is architecture-dependent
  - but not target-dependent
- Pointer-specific operators
  - create: address-of operator (&)
  - access: dereference operator (*)
- Other pointer operators
  - Logical operators (e.g. !, ==, >=)
  - + and – (including +=, ++, etc.)

```
int cyclic = 142857;
double las = 1.303577;
int* pi = NULL;
double* pd = &las;
pi = &cyclic;
*pi = (int) *pd;
```



Illustrate the Point

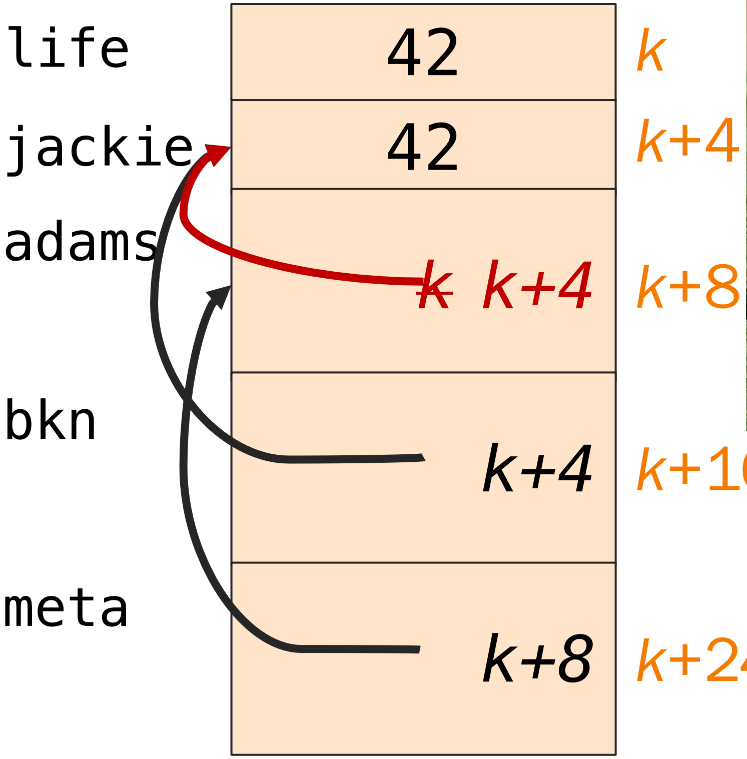
```
int life = 42;
int jackie = 42;
int* adams = &life;
int* bkn = &jackie;
int** meta = &adams;

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

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



```
0 1
1 0 1 0 1
```

I ran out of verbal puns … have an alternate definition

```
adams = bkn;

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

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



```
1 1
1 0 1 1 1
```

ARRAYS

@zburival

Array Design Decisions

Issue: How should C represent arrays?

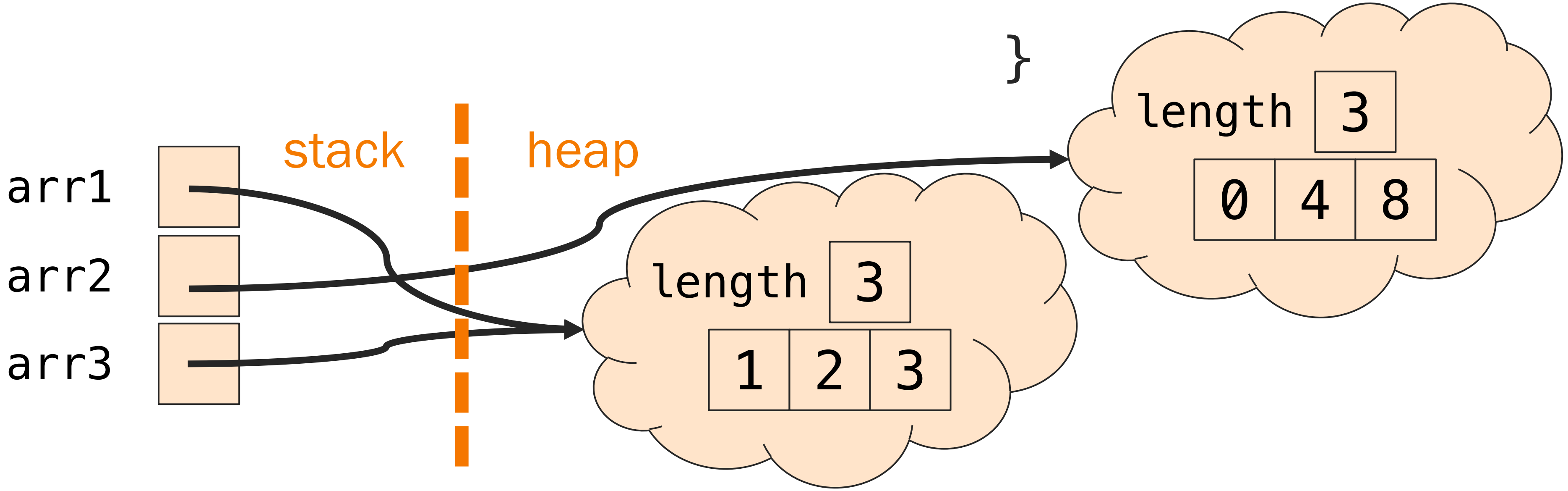
Decision points:

- How to represent collections of elements of the same type?
  - Natural to have a data type corresponding to this
  - Useful to have a single name for the group with iterable naming for individual elements
  - Useful to have them contiguous in memory
- What operations should be possible on arrays?
  - In particular, how to determine length?
- Pass by reference or pass by value?

Refresher: Java Arrays

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

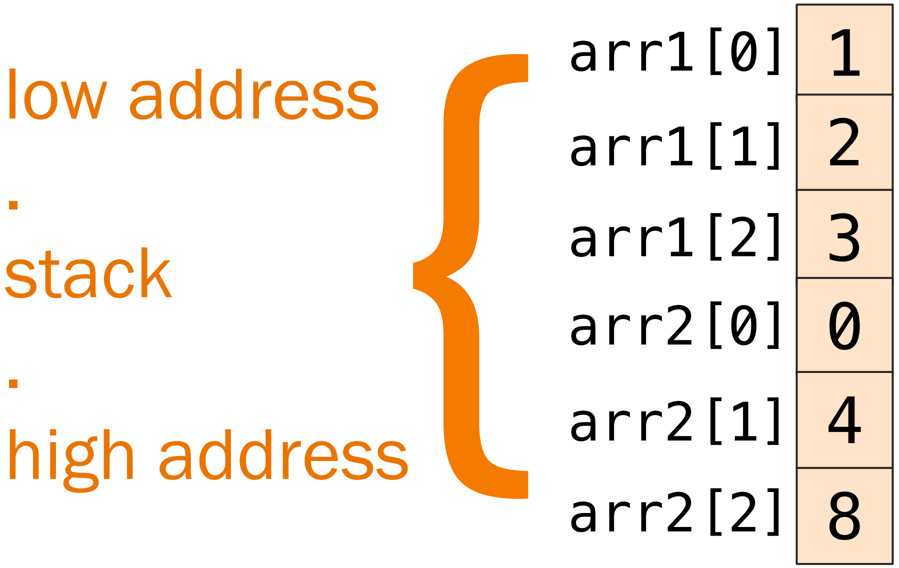
```
public static void arrays() {
  int[] arr1 = {1, 2, 3};
  int[] arr2 = new int[3];
  for(int c = 0;
          c < arr2.length; c++)
      arr2[c] = 4*c;
  int[] arr3 = arr1;
}
```



C Arrays

- Can be statically allocated (in the Stack, BSS, or Data)
  - Length must be known at compile time
- Can also be dynamically allocated (in the Heap)
  - We won’t see this until Lecture 8

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



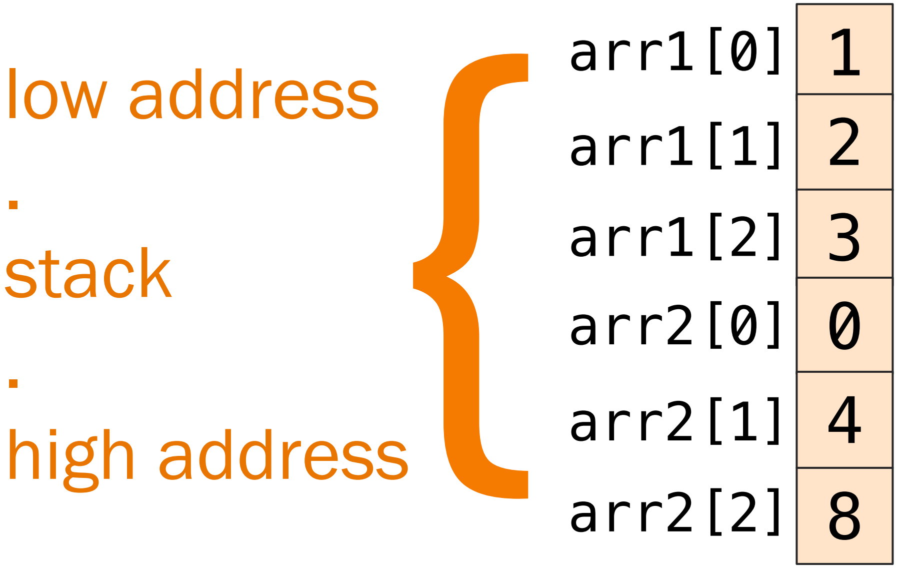
low address
.
stack
.
high address

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

C Arrays

- Can be statically allocated (in the Stack, BSS, or Data)
  - Length must be known at compile time
- Can also be dynamically allocated (in the Heap)
  - We won’t see this until Lecture 8

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



low address

.

stack

.

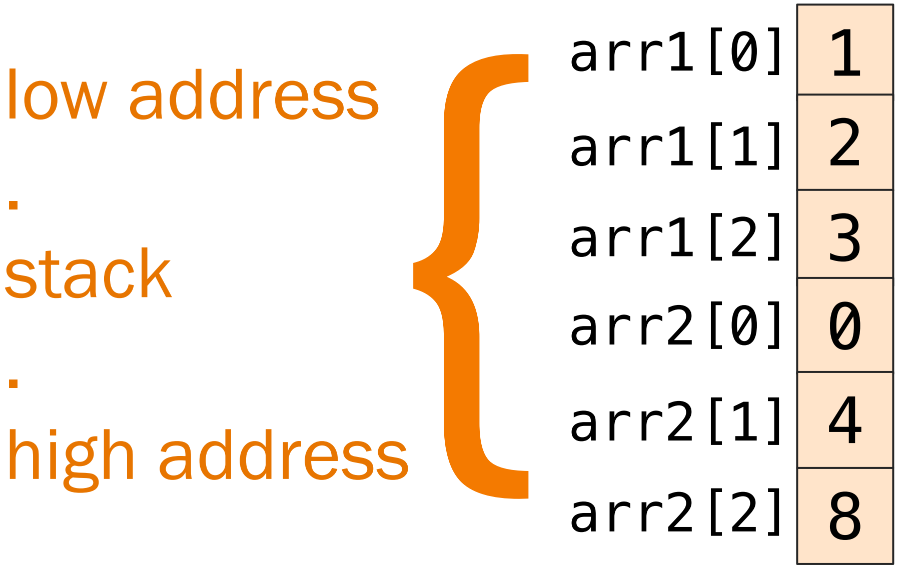
high address

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

C Arrays

- Can be statically allocated (in the Stack, BSS, or Data)
  - Length must be known at compile time
- Can also be dynamically allocated (in the Heap)
  - We won't see this until Lecture 8

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



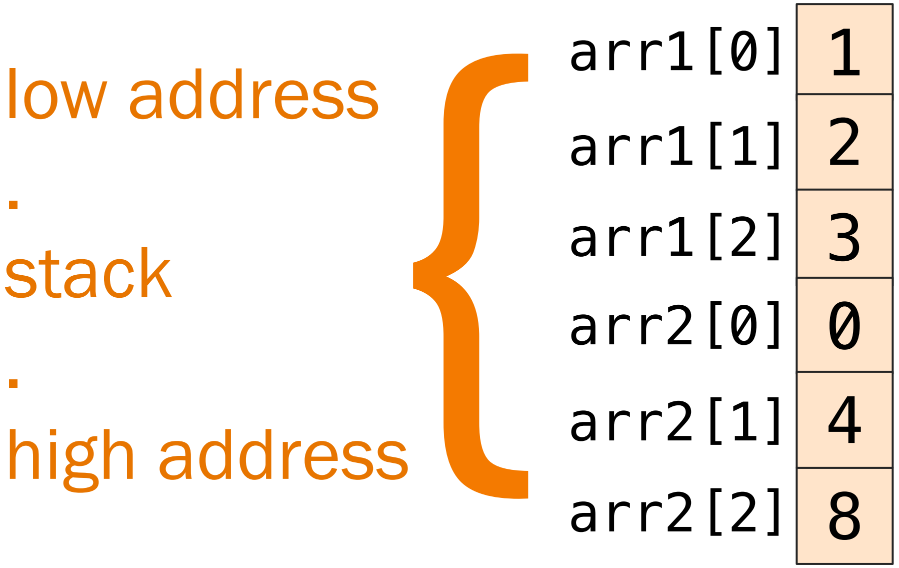
low address
.
stack
.
high address

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

C Arrays

- Can be statically allocated (in the Stack, BSS, or Data)
  - Length must be known at compile time
- Can also be dynamically allocated (in the Heap)
  - We won’t see this until Lecture 8

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



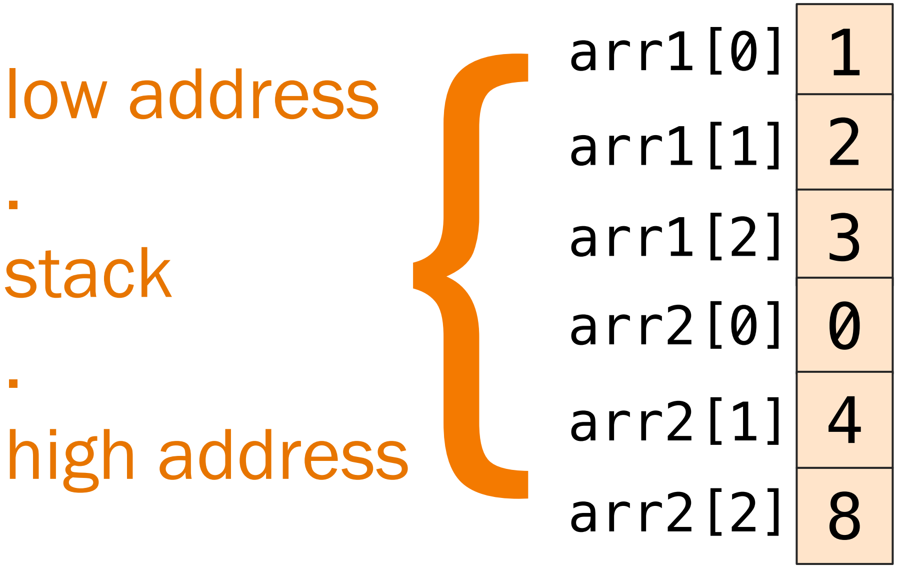
low address
.
stack
.
high address

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

C Arrays

- Can be statically allocated (in the Stack, BSS, or Data)
  - Length must be known at compile time
- Can also be dynamically allocated (in the Heap)
  - We won’t see this until Lecture 8

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

}
```



low address

.

stack

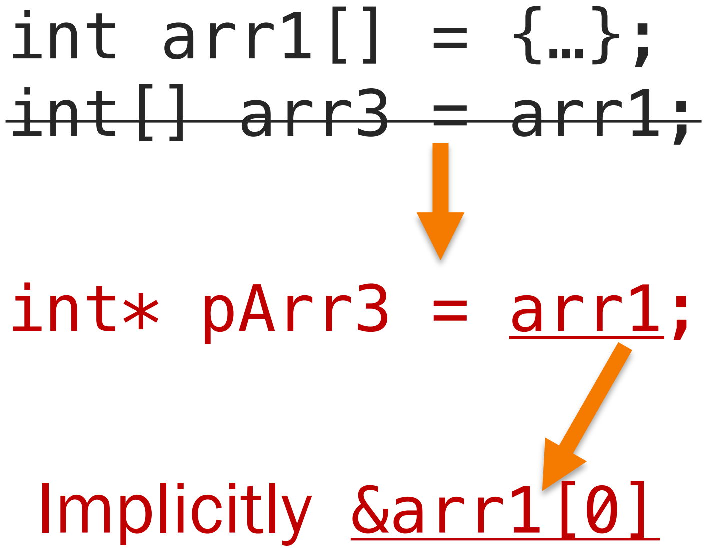
.

high address

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

Pointer/Array Interplay

Array name alone is an implicit pointer:
  `&arr[0]`



`int arr1[] = {…};`

~~int[] arr3 = arr1;~~

`int* pArr3 = arr1;`

Implicitly `&arr1[0]`

Pointer/Array Interplay

Array name alone is an implicit pointer:
`&arr[0]`

Pointers can use the array index operator.

`int arr1[] = {…};`

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

`int* pArr3 = arr1;`

`pArr3[i] = ...;`

Implicitly `&arr1[0]`

Pointer/Array Interplay

Array name alone is an implicit pointer:
`&arr[0]`

Pointers can use the array index operator.

Pointer arithmetic is on elements, not bytes:

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

Array indexing is actually a pointer operation!

`arr[k]` is syntactic sugar for

`*(arr + k)`

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

`int* pArr3 = arr1;`
`pArr3[i] = ...;`

Implicitly `&arr1[0]`

Really `*(pArr3 + i)`

Arrays with Functions

- Pass 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”
- Return 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 from the function’s stack!

```
/* completely equivalent
  function signatures */
size_t count(int numbers[]);
size_t count(int* numbers);
size_t count(int numbers[5]);
/* always 8 */
return sizeof(numbers);
```

```
~~int[] getArr();~~
int* getArr();
```

STRINGS

String Design Decisions

Issue: How should C represent strings and string literals?

Decision Points:

- Natural to represent a string as a sequence of contiguous chars
  - Even if we just saw how chars can be insufficient
- How to know where char sequence ends?
  - Store length together with char sequence?
  - Store special “sentinel” char after char sequence?

Strings and String Literals

Decisions

- Adopt a convention
  - String is a sequence of contiguous chars
  - String is terminated with null char (‘\0’)
- Use double-quote syntax (e.g., "hello") to represent a string literal
  - Allow string literals to be used as special-case initializer lists
- Provide 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?

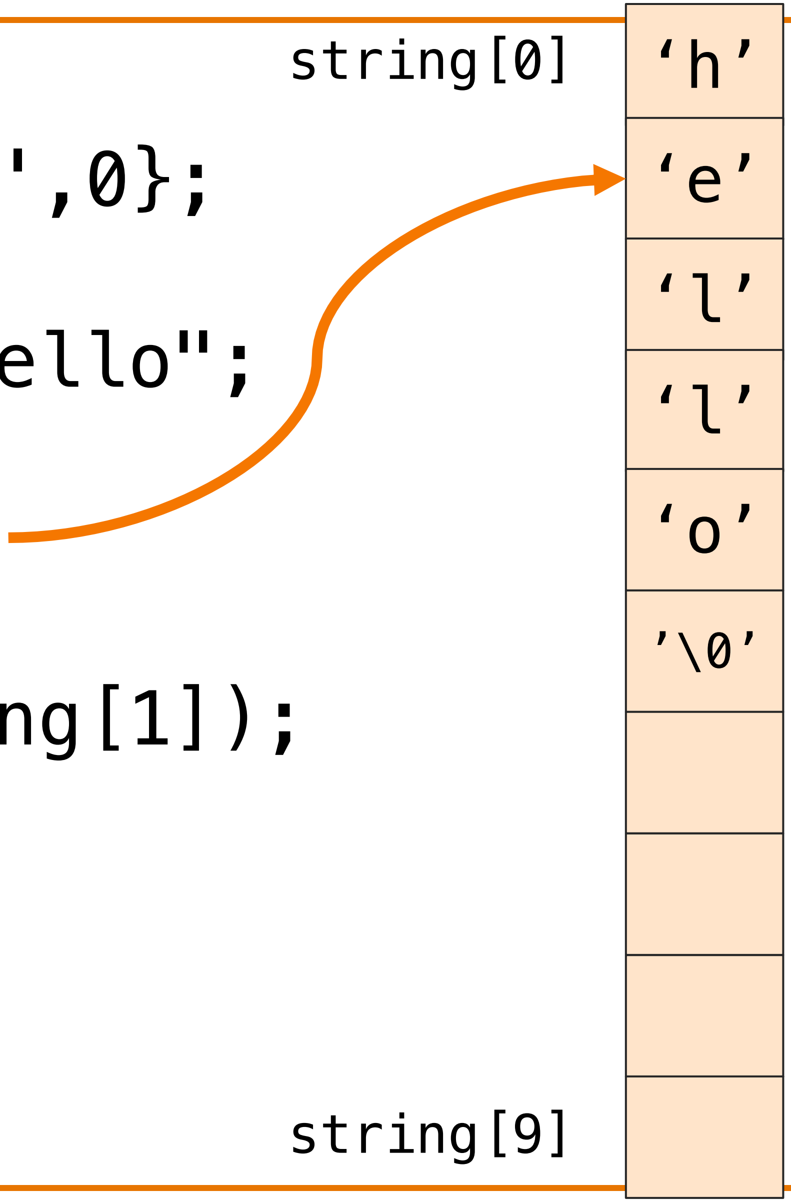
What decisions did the designers of Java make?

Lemon Gelatin Dessert

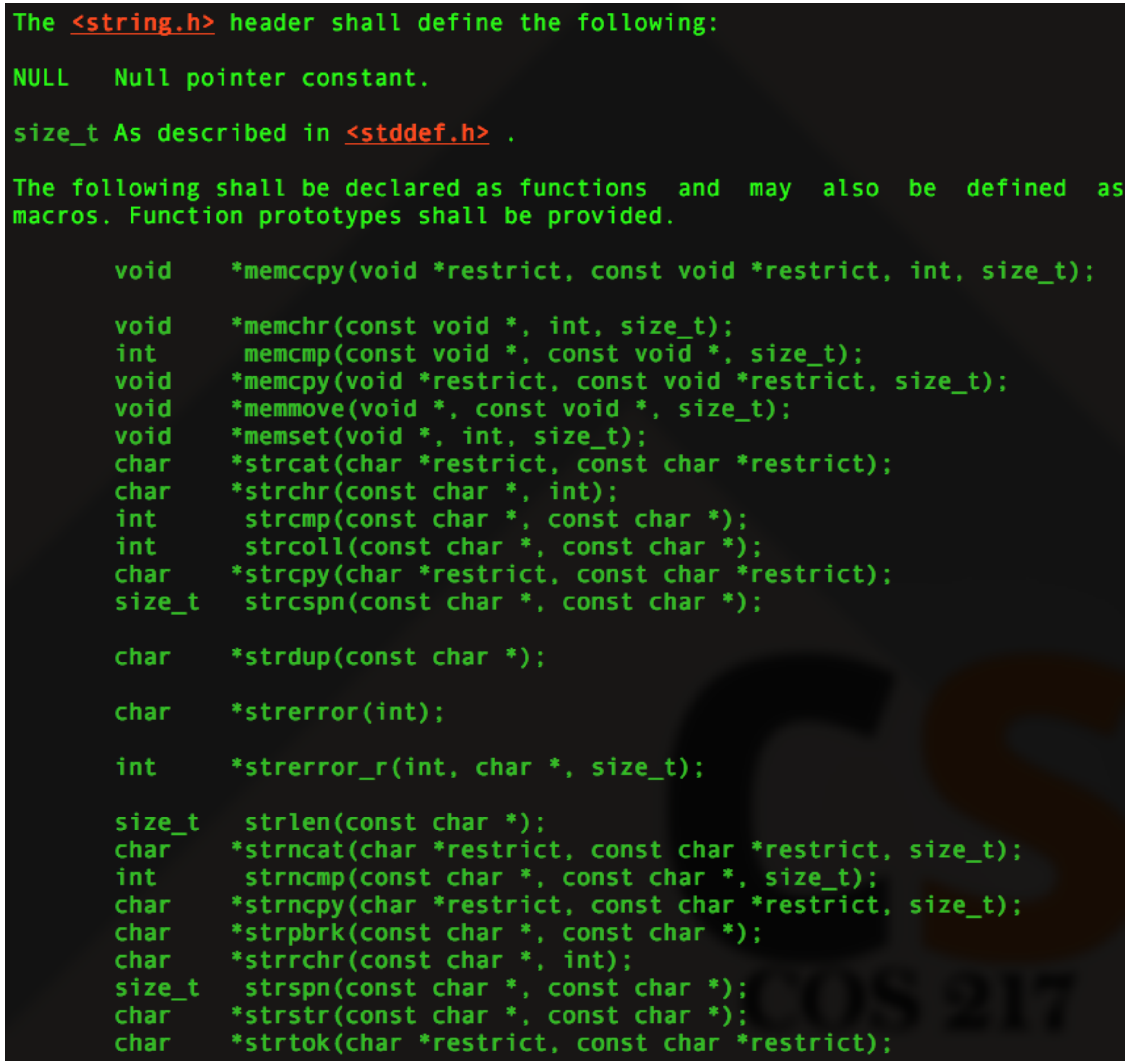
```
char string[10] =
 {'H','e','l','l','o',0};
```

(or, equivalently)

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

char* pc = string+1;

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



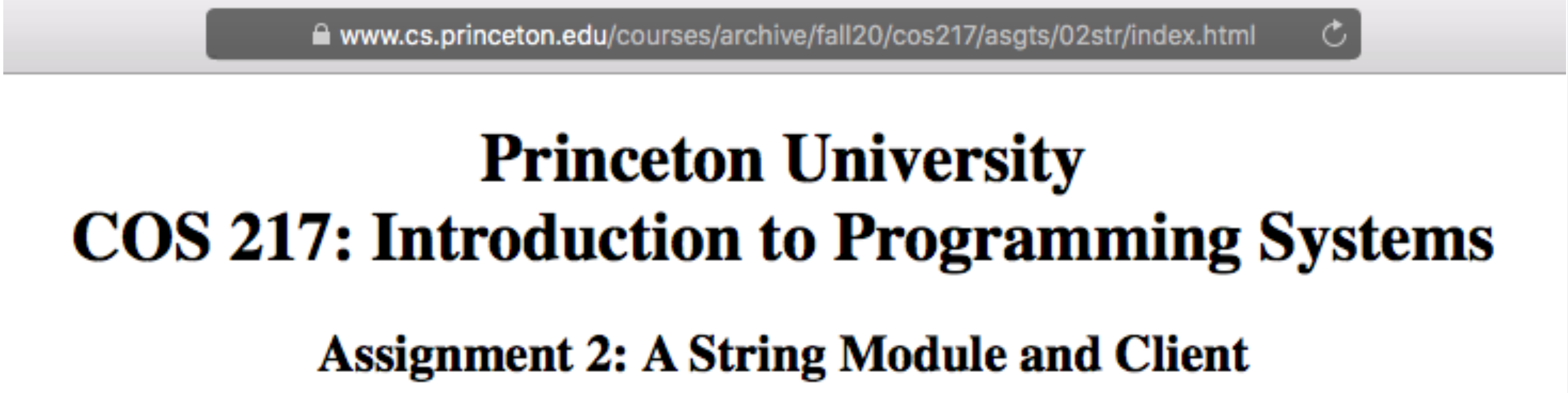
Standard String Library



```
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];
  int 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)



www.cs.princeton.edu/courses/archive/fall20/cos217/asgts/02str/index.html

Princeton University
COS 217: Introduction to Programming Systems

Assignment 2: A String Module and Client