CS 110
Computer Architecture

Lecture 3: *Introduction to C II*

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Agenda

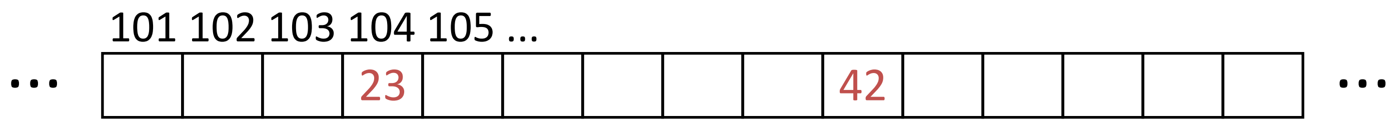
- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

Agenda

- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

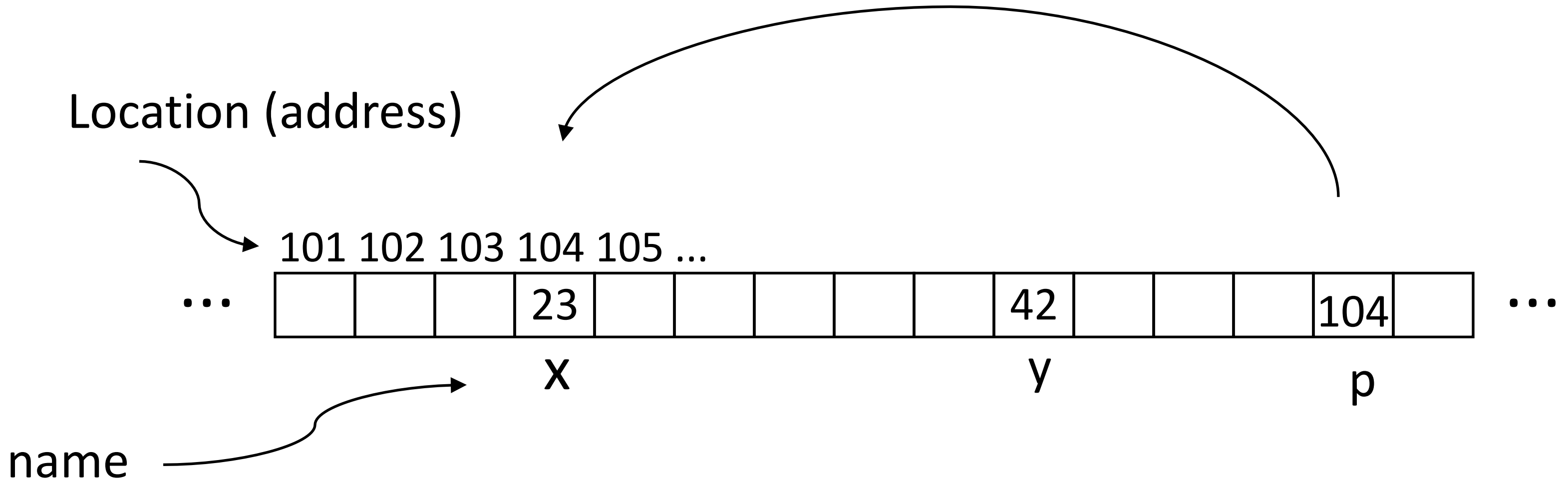
Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - For addresses do we use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored there



Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



Pointer Syntax

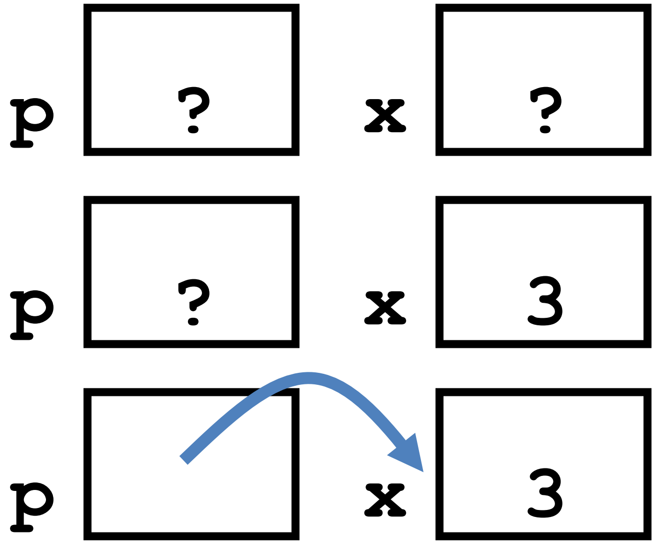
- `int *x;`
  - Tells compiler that variable x is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the “dereference operator” in this context

Creating and Using Pointers

- How to create a pointer:

  & operator: get address of a variable

```
int *p, x;

    x = 3;

    p = &x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

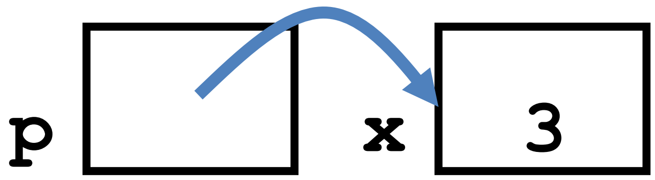
- How get a value pointed to?

  "*" (dereference operator): get the value that the pointer points to

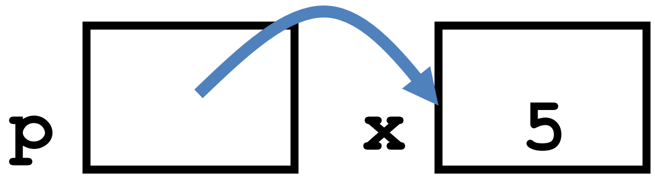
```
printf("p points to value %d\n",*p);
```

Using Pointer for Writes

- How to change a variable pointed to?
  - Use the dereference operator `*` on left of assignment operator `=`



```
*p = 5;
```



Pointers and Parameter Passing

- C passes parameters “by value”
  - Procedure/function/method gets a copy of the parameter, *so changing the copy cannot change the original*

```
void add_one (int x) {
    x = x + 1;
  }
int y = 3;
add_one(y);
```

y remains equal to 3

Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

```
void add_one (int *p) {
   *p = *p + 1;
  }
int y = 3;

add_one(&y);
```

y is now equal to 4

What would you use in C++?

Call by reference:

```
void add_one (int &p) {
   p = p + 1;   // or  p += 1;
}
```

Types of Pointers

- Pointers are used to point to any kind of data (**`int`**, **`char`**, a **`struct`**, etc.)
- Normally a pointer only points to one type (**`int`**, **`char`**, a **`struct`**, etc.).
  - **`void *`** is a type that can point to anything (generic pointer)
  - Use **`void *`** sparingly to help avoid program bugs, and security issues, and other bad things!

More C Pointer Dangers

- *Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!*
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?

```
void f()
{
     int *ptr;
     *ptr = 5;
}
```

Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
```

Note: C structure assignment is not a ”deep copy”. All members are copied, but not things pointed to by members.

Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—coming up next week
    - *Dangling references* and *memory leaks*

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 100,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers in application code
- Low-level system code still needs low-level access via pointers

Agenda

- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

C Arrays

- Declaration:

```
int ar[2];
```

declares a 2-element integer array: just a block of memory

```
int ar[] = {795, 635};
```

declares and initializes a 2-element integer array

C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (aka “null terminator”)

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```

Array Name / Pointer Duality

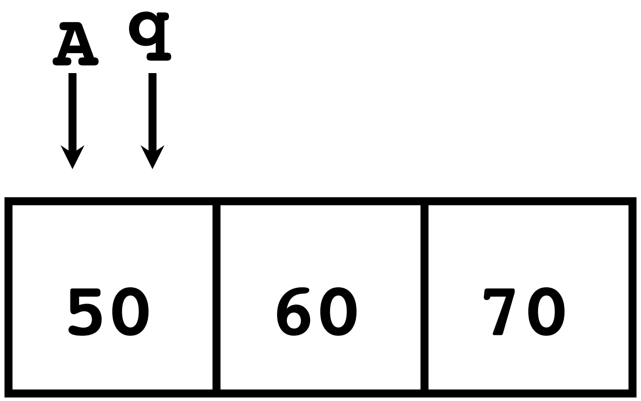
- *Key Concept*: Array variable is a “pointer” to the first (0th) element
- So, array variables almost identical to pointers
  - **`char *string`** and **`char string[]`** are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays, sizeof
- Consequences:
  - **`ar`** is an array variable, but works like a pointer
  - **`ar[0]`** is the same as **`*ar`**
  - **`ar[2]`** is the same as **`*(ar+2)`**
  - Can use pointer arithmetic to conveniently access arrays

Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

```
void inc_ptr(int *p)
{    p =  p + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr( q);
printf("*q = %d\n", *q);
```

```
*q = 50
```

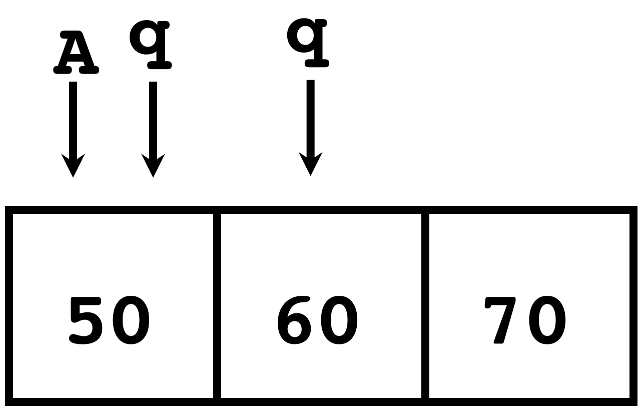


Pointer to a Pointer

- Solution! Pass a pointer to a pointer, declared as `**h`
- Now what gets printed?

```
void inc_ptr(int **h)
{    *h = *h + 1;    }

int A[3] = {50, 60, 70};
int *q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```

```
*q = 60
```



C Arrays are Very Primitive

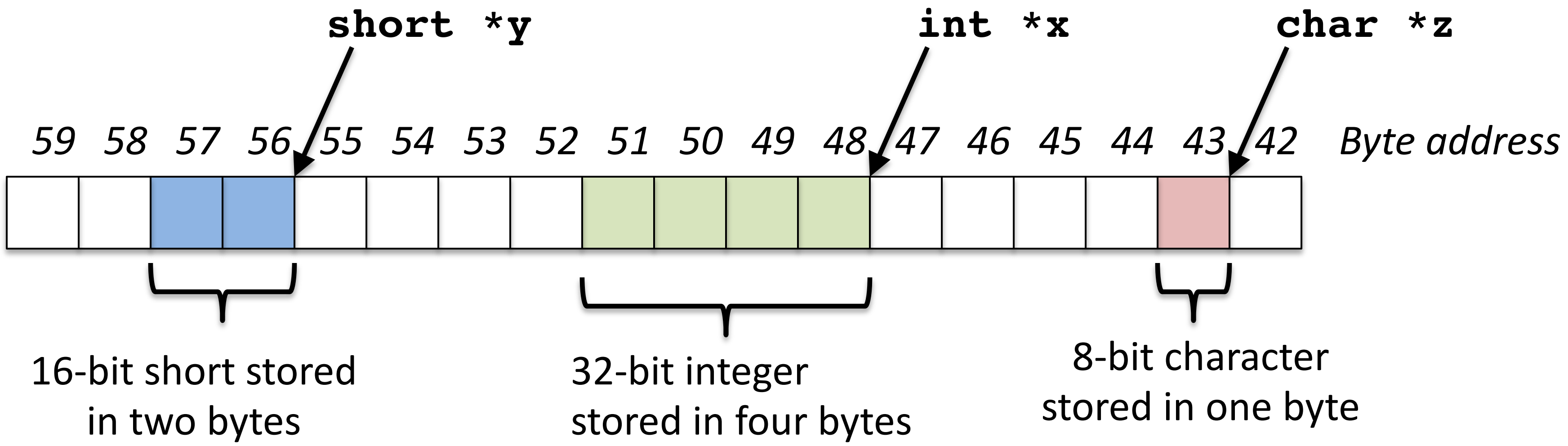
- An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array *and its size* to any procedure that is going to manipulate it
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful!

Use Defined Constants

- Array size *n*; want to access from *0* to *n-1*, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: "Don't Repeat Yourself"

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware's memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes


short *y
int *x
char *z
59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42
Byte address
16-bit short stored in two bytes
32-bit integer stored in four bytes
8-bit character stored in one byte

sizeof() operator

- sizeof(type) returns number of bytes in object
  - But number of bits in a byte is not standardized
    - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
- By definition, sizeof(char)==1
- Can take sizeof(arr), or sizeof(structtype)
- We'll see more of sizeof when we look at dynamic memory management

Pointer Arithmetic

pointer + number *pointer – number*

e.g., *pointer* **+ 1** adds 1 <u>something</u> to a pointer

```
char    *p;
char     a;
char     b;

p = &a;
p += 1;
```

```
int     *p;
int      a;
int      b;

p = &a;
p += 1;
```

In each, p now points to b (Assuming compiler doesn't reorder variables in memory. ***Never code like this!!!!***)

Adds **`1*sizeof(char)`** to the memory address

Adds **`1*sizeof(int)`** to the memory address

Pointer arithmetic should be used <u>cautiously</u>

Arrays and Pointers

- Array ≈ pointer to the initial (0th) array element

`a[i] ≡ *(a+i)`

- An array is passed to a function as a pointer
  - The array size is lost!
- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
   printf("%d\n", sizeof(a));

}
```

What does this print (32bit)? **4**

... because **`array`** is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print (32bit)? **40**

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These code sequences have the same effect!

C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (aka “null terminator”)

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```

Concise strlen()

```
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        ; /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?

Point past end of array?

- Array size n; want to access from 0 to $n-1$, but test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
 ...
 p = &ar[0]; q = &ar[10];
 while (p != q)
     /* sum = sum + *p; p = p + 1; */
  sum += *p++;
```

  – Is this legal?

- C defines that one element past end of array must be a valid address, i.e., not cause an error

Valid Pointer Arithmetic

- Add an integer to a pointer.
- Subtract 2 pointers (in the same array)
- Compare pointers (<, <=, ==, !=, >, >=)
- Compare pointer to NULL (indicates that the pointer points to nothing)

Everything else illegal since makes no sense:

- adding two pointers
- multiplying pointers
- subtract pointer from integer

Arguments in `main()`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[])`
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:

    unix% sort myFile

  - `argv` is a *pointer* to an array containing the arguments as strings

Example

- `foo hello 87`
- `argc = 3 /* number arguments */`

-
  ```
  argv[0] = "foo",
  argv[1] = "hello",
  argv[2] = "87"
  ```
  - Array of pointers to strings

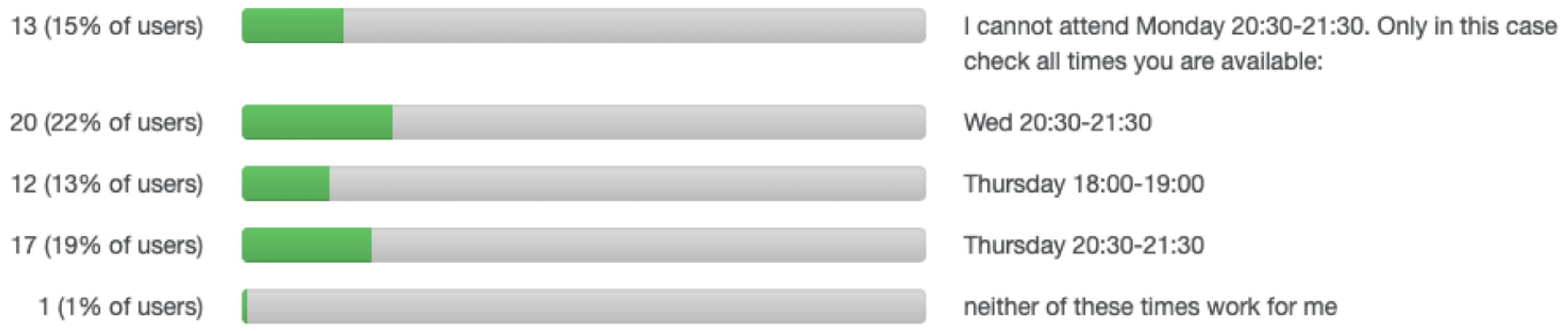
Summary

- Pointers and arrays are virtually same
- C knows how to increment pointers
- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized
- (Beware) The cost of efficiency is more overhead for the programmer.
  - "C gives you a lot of extra rope but be careful not to hang yourself with it!"

ADMIN

Admin

- HW 1 due tomorrow!!!
- Discussion session today:
  - Monday 20:30-21:30. 教学中心 201
  - 2nd: Wednesday 20:30-21:30 ?
    But how about the “Situation and Policy” lectures?



Leaving Early?

Agenda

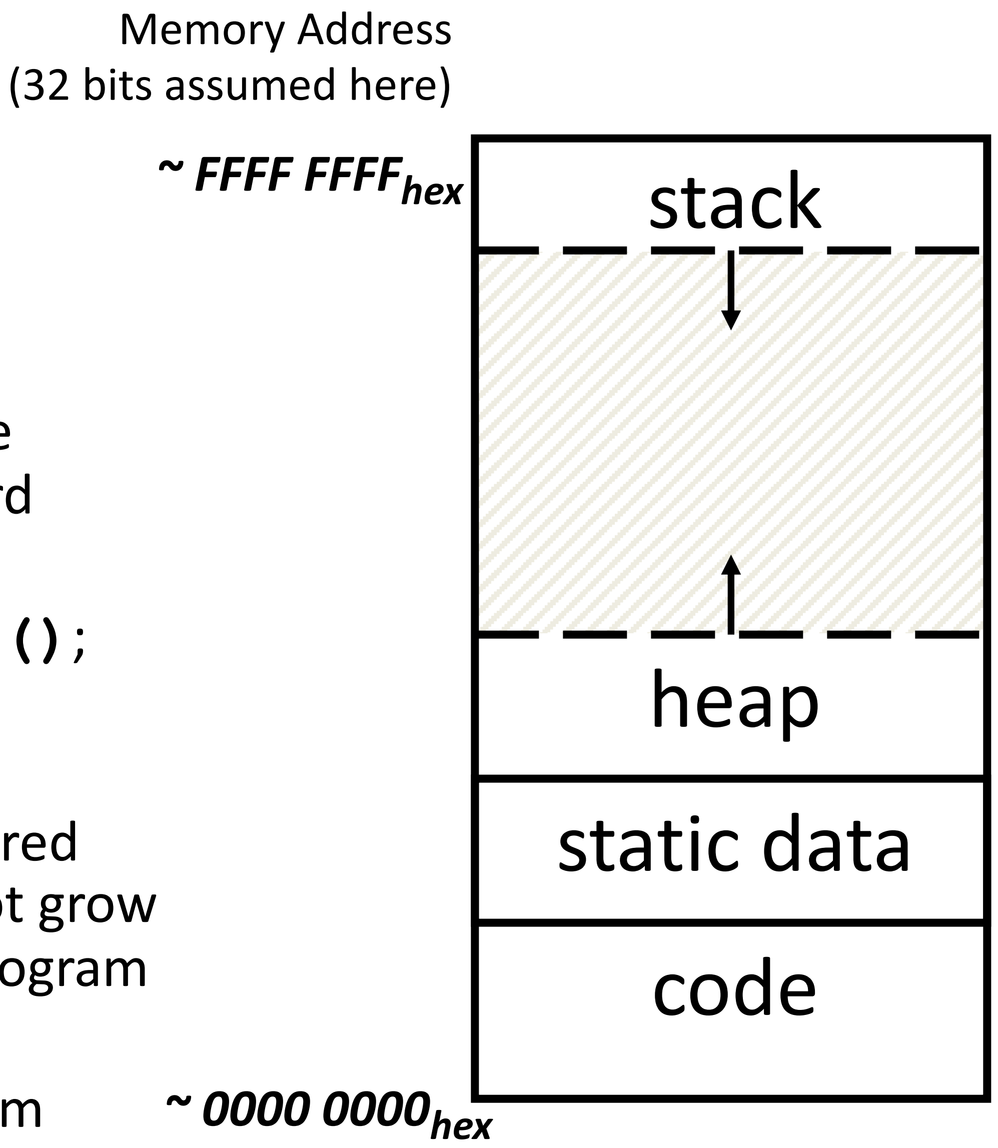
- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

C Memory Management

- How does the C compiler determine where to put all the variables in machine's memory?
- How to create dynamically sized objects?
- To simplify discussion, we assume one program runs at a time, with access to all of memory.
- Later, we'll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory.

C Memory Management

- Program's *address space* contains 4 regions:
  - stack: local variables inside functions, grows downward
  - heap: space requested for dynamic data via **`malloc()`**; resizes dynamically, grows upward
  - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code: loaded when program starts, does not change



Where are Variables Allocated?

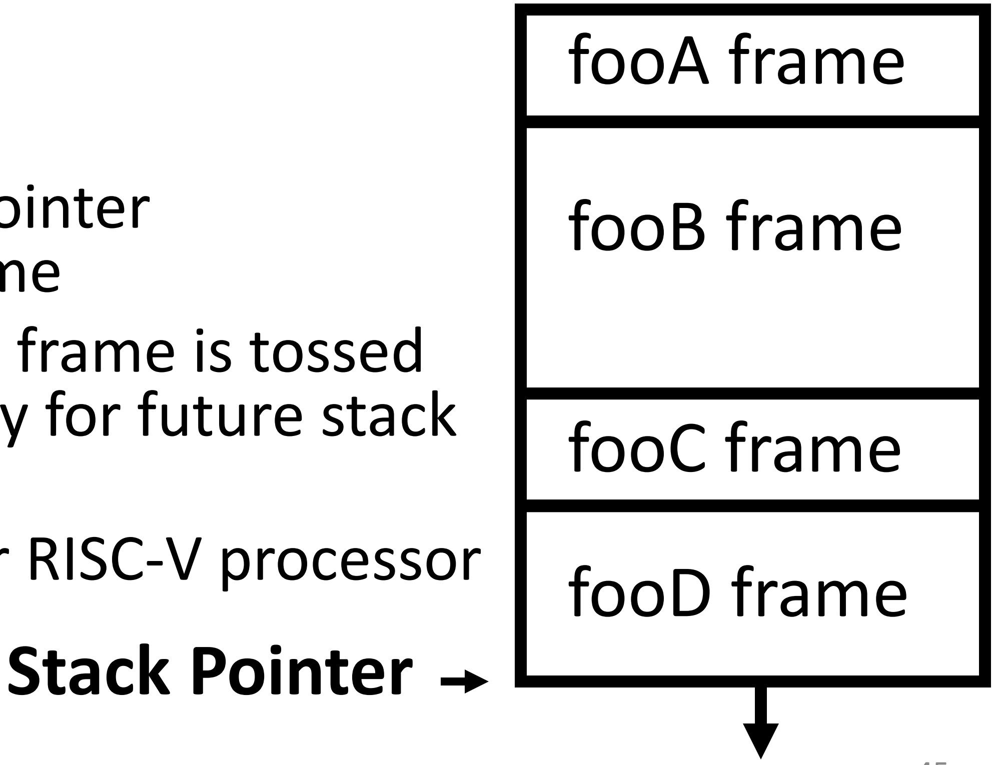
- If declared outside a function, allocated in “static” storage
- If declared inside function, allocated on the “stack” and freed when function returns
  - main() is treated like a function

```
int myGlobal;
main() {
   int myTemp;
}
```

The Stack

- Every time a function is called, a new frame is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack frame is tossed off the stack; frees memory for future stack frames
- We'll cover details later for RISC-V processor

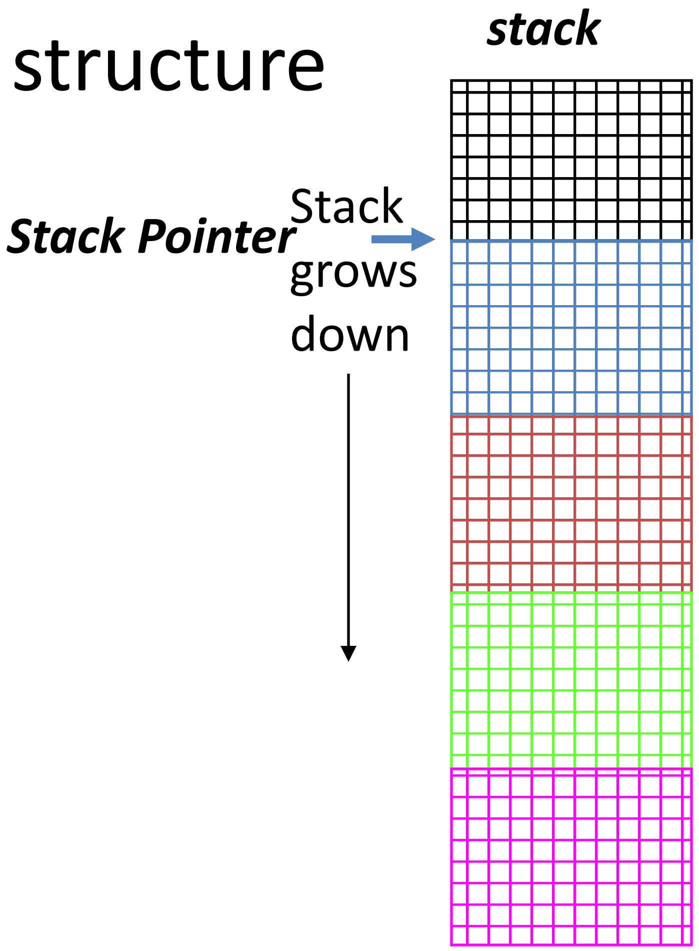
```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```



Stack Animation

- Last In, First Out (LIFO) data structure

```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }void d (int p)
         {
         }
```



Managing the Heap

C supports five functions for heap management:

- **`malloc()`** allocate a block of uninitialized memory
- **`calloc()`** allocate a block of zeroed memory
- **`free()`** free previously allocated block of memory
- **`realloc()`** change size of previously allocated block
  - careful – it might move!

Malloc()

- **`void *malloc(size_t n)`**:
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls might not yield blocks in contiguous addresses
  - **`n`** is an integer, indicating size of allocated memory block in bytes
  - **`size_t`** is an unsigned integer type big enough to "count" memory bytes
  - **`sizeof`** returns size of given type in bytes, produces more portable code
  - Returns **`void*`** pointer to block; **`NULL`** return indicates no more memory
  - Think of pointer as a *handle* that describes the allocated block of memory; Additional control information stored in the heap around the allocated block!

- Examples:

"Cast" operation, changes type of a variable.
Here changes ***`(void *)`*** *to* ***`(int *)`***

```
int *ip;
ip = (int *) malloc(sizeof(int));

typedef struct { … } TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
```

Managing the Heap

- **`void free(void *p):`**
  - Releases memory allocated by **`malloc()`**
  - **`p`** is pointer containing the address *originally* returned by **`malloc()`**

```
int *ip;
ip = (int *) malloc(sizeof(int));
... .. ..
free((void*) ip);  /* Can you free(ip) after ip++ ? */

typedef struct {… } TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
    ... .. ..
free((void *) tp);
```

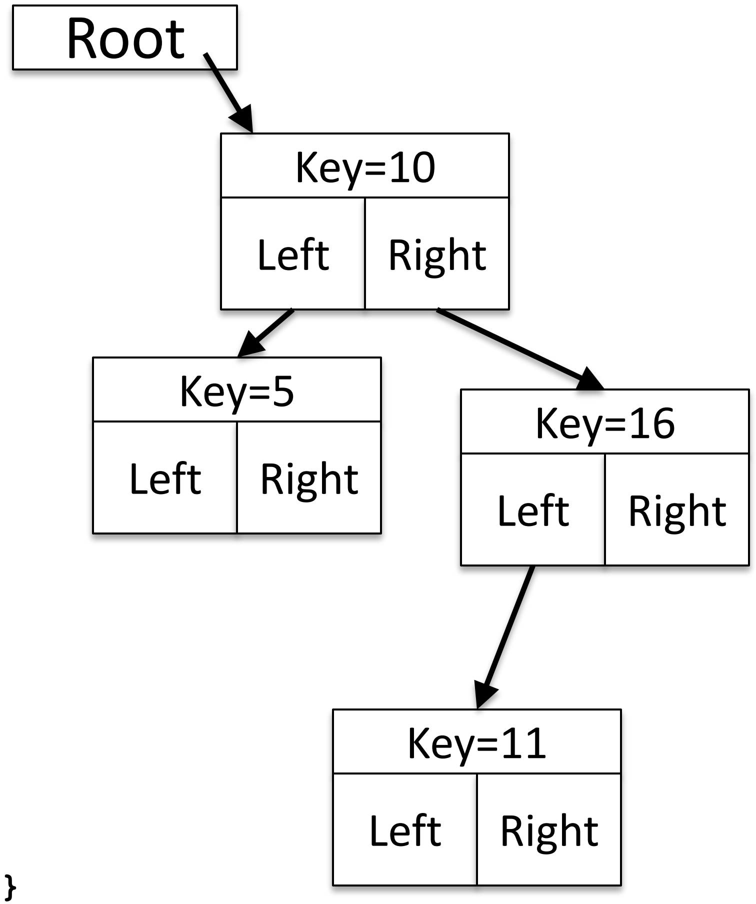
  - When insufficient free memory, **`malloc()`** returns **`NULL`** pointer; **Check for it!**

```
if ((ip = (int *) malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1); /* Crash and burn! */
}
```

  - When you free memory, you must be sure that you pass the **original address** returned from **`malloc()`** to **`free()`**; Otherwise, system exception (or worse)!

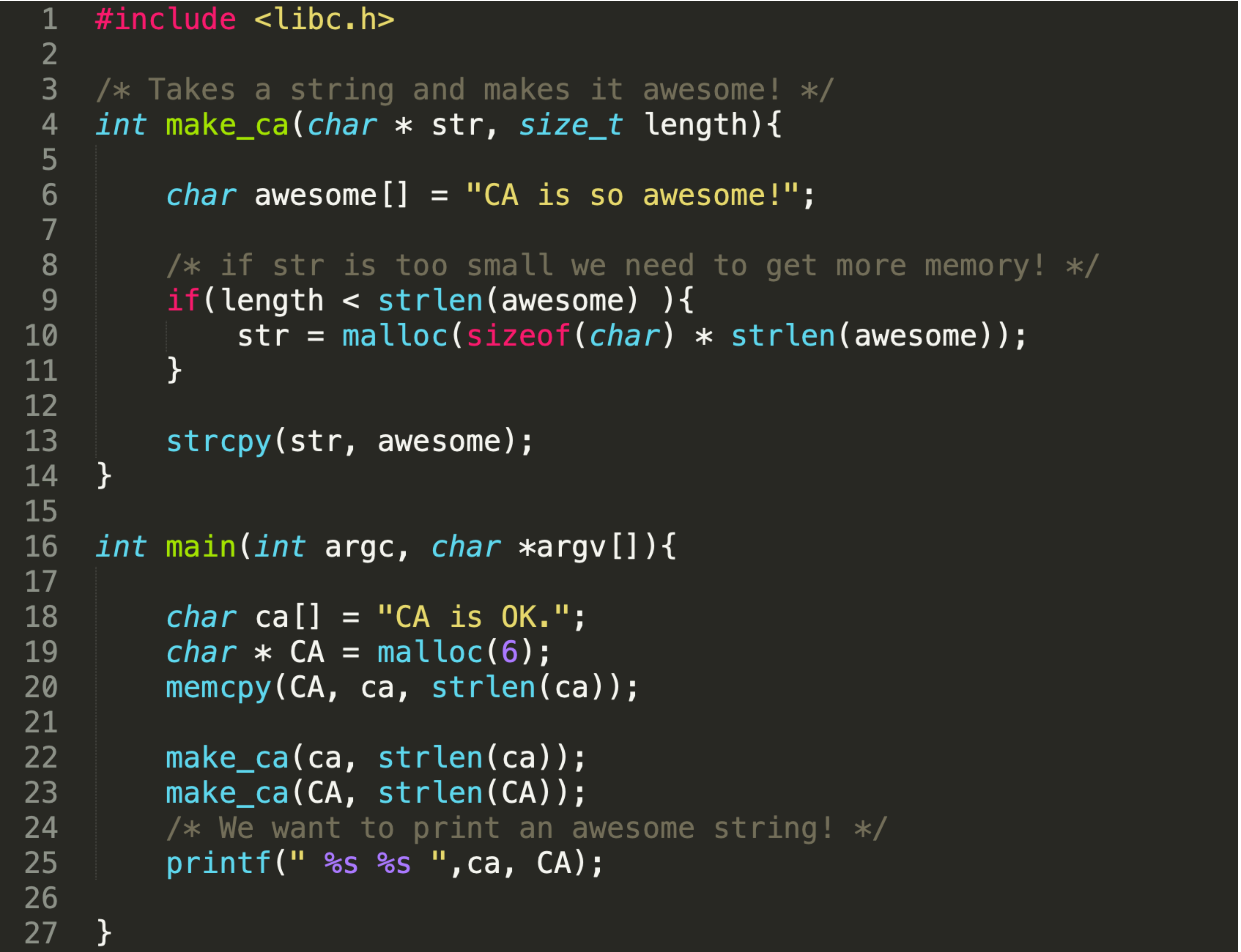
Using Dynamic Memory

```
typedef struct node {
        int key;
        struct node *left;
        struct node *right;
} Node;

Node *root = 0;

Node *create_node(int key, Node *left, Node *right)
{
   Node *np;
   if ( (np = (Node*) malloc(sizeof(Node))) == NULL)
   { printf("Memory exhausted!\n"); exit(1); }
   else
   {  np->key = key;
      np->left = left;
      np->right = right;
      return np;
   }
}

void insert(int key, Node **tree)
{
   if ( (*tree) == NULL)
   { (*tree) = create_node(key, NULL, NULL); return; }

   if (key <= (*tree)->key)
      insert(key, &((*tree)->left));
   else
      insert(key, &((*tree)->right));
}
```



Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time



```c
#include <libc.h>

/* Takes a string and makes it awesome! */
int make_ca(char * str, size_t length){

    char awesome[] = "CA is so awesome!";

    /* if str is too small we need to get more memory! */
    if(length < strlen(awesome) ){
        str = malloc(sizeof(char) * strlen(awesome));
    }

    strcpy(str, awesome);
}

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

    char ca[] = "CA is OK.";
    char * CA = malloc(6);
    memcpy(CA, ca, strlen(ca));

    make_ca(ca, strlen(ca));
    make_ca(CA, strlen(CA));
    /* We want to print an awesome string! */
    printf(" %s %s ",ca, CA);

}
```

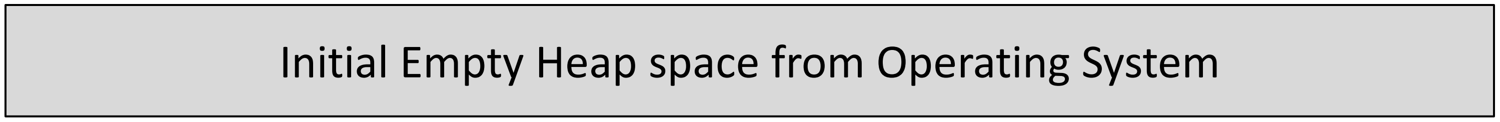
Bugs

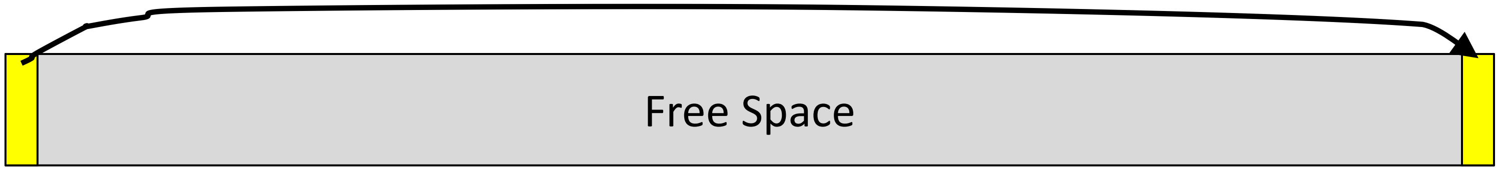
- Line 9: comparison with strlen instead of sizeof (for 0-terminator)
- Line 10: strlen instead of sizeof (or +1) for malloc =>
  - Line 13: write past end of array (if malloc was used)
- Line 4: Ownership of pointer str not clear =>
  - Line 10: Potential memory leak
- Line 4: New pointer is not returned/ no pointer to pointer is used
- Line 20: memcpy over length of CA
- Line 20: 0-terminator is not copied!
- Line 22 &23: better: call with array size
- Line 14 & 27: return missing!

How are Malloc/Free implemented?

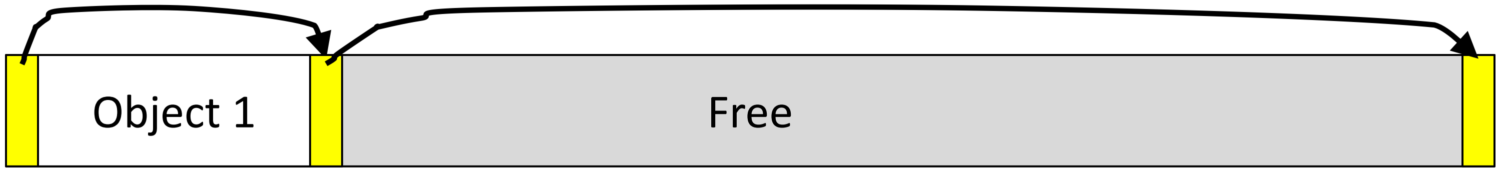
- Underlying operating system allows **`malloc`** library to ask for large blocks of memory to use in heap (e.g., using Unix **`sbrk()`** call)
- C standard **`malloc`** library creates data structure inside unused portions to track free space

Simple Slow Malloc Implementation

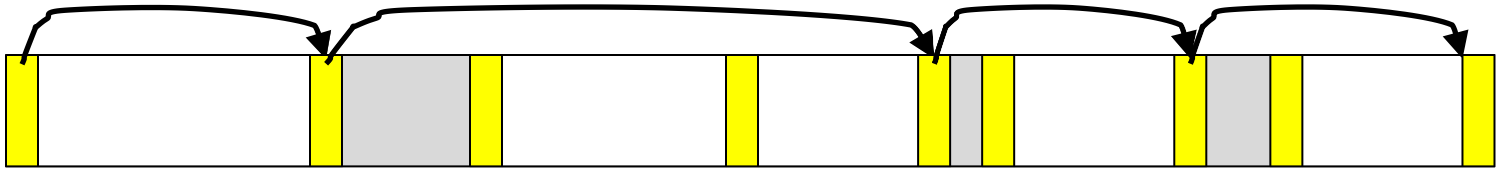




Malloc library creates linked list of empty blocks (one block initially)



First allocation chews up space from start of free space

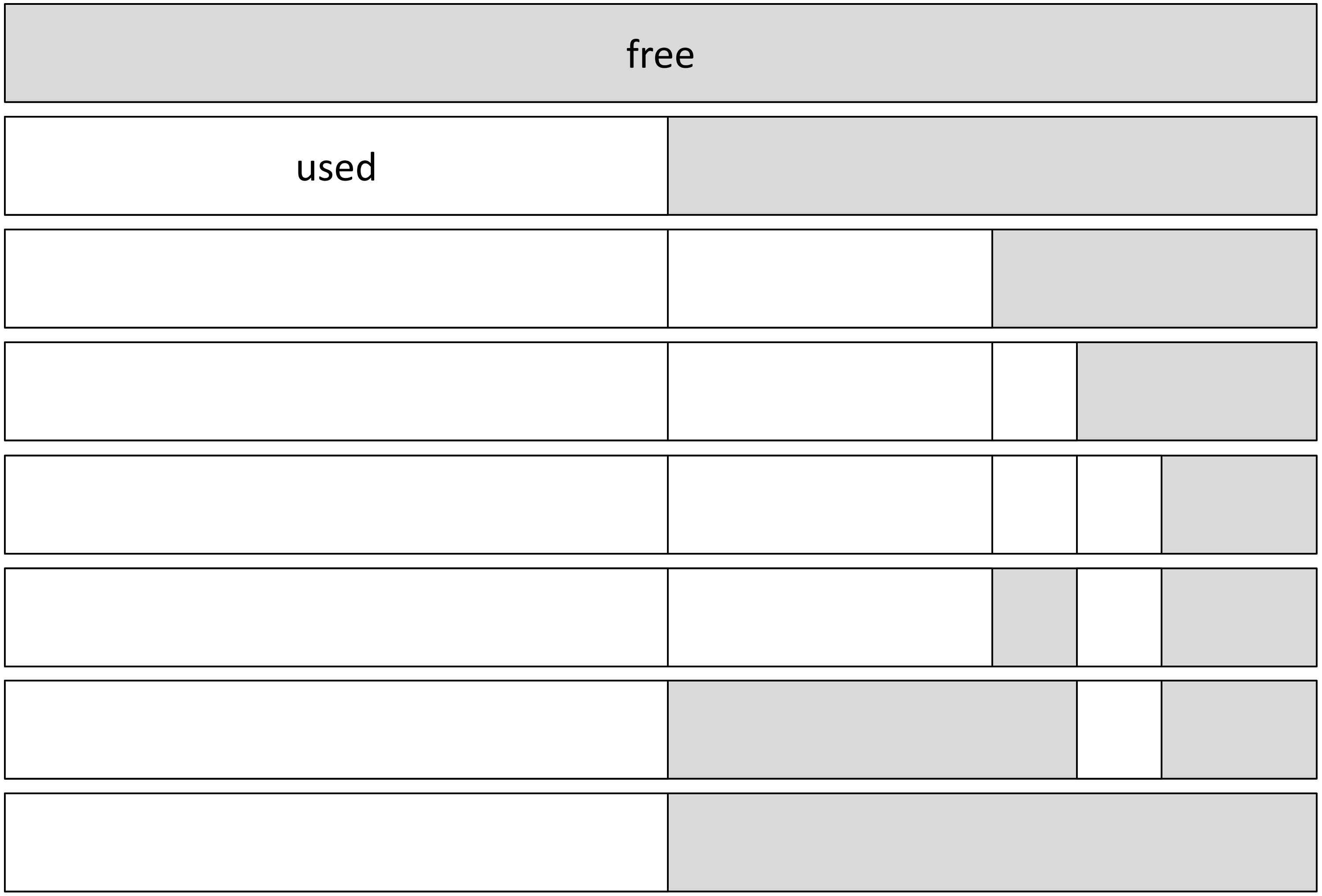


After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:

Power-of-2 “Buddy Allocator”



Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and space in unallocated memory to hold **`malloc`**'s internal data structures
- Rely on programmer remembering to free with same pointer returned by **`malloc`**
- Rely on programmer not messing with internal data structures accidentally!

Agenda

- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Using Memory You Don't Own

- What is wrong with this code?
- Using pointers beyond the range that had been malloc'd
    - –May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
     int i, j;
     ipr = (*int) malloc(4 * sizeof(int));
      i = *(ipr - 1000); j = *(ipr + 1000);
     free(ipr);
   }

   void WriteMem() {
     ipw = (*int) malloc(5 * sizeof(int));
     *(ipw - 1000) = 0; *(ipw + 1000) = 0;
     free(ipw);
   }
```

Faulty Heap Management

- What is wrong with this code?

```
int *pi;
void foo() {
   pi = malloc(8*sizeof(int));
   …
   free(pi);
}

void main() {
   pi = malloc(4*sizeof(int));
   foo();
   …
}
```

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
   pi = malloc(8*sizeof(int));
   /* Allocate memory for pi */
   /* Oops, leaked the old memory pointed to by pi */
   …
   free(pi); /* foo() is done with pi, so free it */
}

void main() {
   pi = malloc(4*sizeof(int));
   foo(); /* Memory leak: foo leaks it */
  …
}
```

Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
  … … …
   plk++;
}
```

Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
  … … …
   plk++;
}
```

Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
    int fnh = 0;
    free(&fnh);
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    free(fum);
    free(fum);
}
```

Faulty Heap Management

- Can’t free non-heap memory; Can’t free memory that hasn’t been allocated

```
void FreeMemX() {
   int fnh = 0;
   free(&fnh);
}

void FreeMemY() {
   int *fum = malloc(4 * sizeof(int));
   free(fum+1);
   free(fum);
   free(fum);
}
```

Using Memory You Haven’t Allocated

- What is wrong with this code?

```
void StringManipulate() {
   const char *name = “Safety Critical";
   char *str = malloc(10);
   strncpy(str, name, 10);
   str[10] = '\0';
   printf("%s\n", str);
}
```

Using Memory You Haven’t Allocated

- Reference beyond array bounds

```
void StringManipulate() {
   const char *name = “Safety Critical";
   char *str = malloc(10);
   strncpy(str, name, 10);
   str[10] = '\0';
  /* Write Beyond Array Bounds */
   printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```

Using Memory You Don’t Own

- What’s wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
        struct node* next;
        int val;
    } Node;

    int findLastNodeValue(Node* head) {
        while (head->next != NULL) {
            head = head->next;
        }
        return head->val;
    }
```

Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

```
typedef struct node {
        struct node* next;
        int val;
    } Node;

    int findLastNodeValue(Node* head) {
        while (head->next != NULL) {
             head = head->next;
        }
        return head->val;
    }
```

Managing the Heap

- `realloc(p,size)`:
  - Resize a previously allocated block at `p` to a new `size`
  - If `p` is `NULL`, then `realloc` behaves like `malloc`
  - If `size` is 0, then `realloc` behaves like `free`, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don’t Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}
```

Remember: `realloc` may move entire block

```
int* fill_fibonacci(int *fib, int size) {
   int i;
   /* oops, forgot: fib = */ init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

What if array is moved to new location?

And In Conclusion, …

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - `*` “follows” a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers
- C is an efficient language, but leaves safety to the programmer
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code