信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture

C Memory Management

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2024/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2024/3/7

Administrative

- HW2 and Lab2 (with guidance to valgrind) will be released, keep an eye on Piazza
- Start early on labs so that checking can be faster
- Proj1.1 will be released on Piazza next week
- Those who haven't enrolled in Piazza and Gradescope please enroll yourselves ASAP!
- Discussion next week on memory management & valgrind (useful for your labs/HWs/projects) by TA Suting Chen at teaching center 301 on Monday
  - Discussion on Mar. 15th will be held at SPST 4-122 (only this one)
- The similarity check is conducted automatically for each HW, proj, etc. Please comply with the course rules!
- We grade only on your most recent submissions before ddl for all assignments!

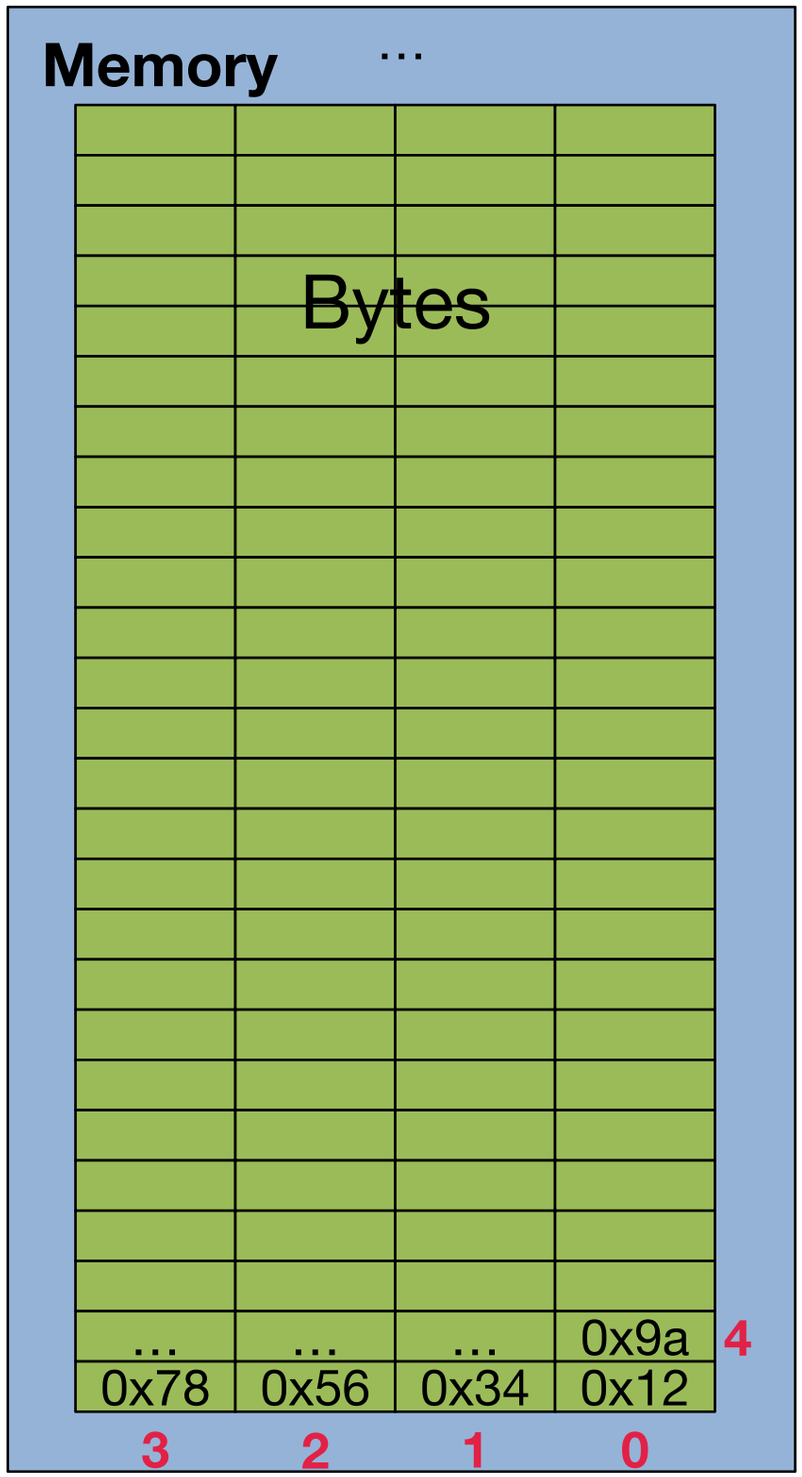
Outline

- Pointer
- Array
- Pointer arithmetic
- C memory management
  - Stack
  - Heap

Address vs. Value

- Consider memory to be a single huge array of bytes
  - Each cell/byte of the array has an address associated with it
  - Each cell also stores some value
- Don't confuse the address referring to a memory location with the value stored there
- Byte-addressable
  - `int`: 4 bytes (by convention)
  - `char`: 1 byte (C standard)
  - address itself: machine-dependent

  (4 bytes throughout this course, 4 GB memory)
  - RV32I instructions: 4 bytes



Pointers

- An address refers to a particular memory location;
- Pointer: A variable that contains the address of a variable;
- Pointer syntax and basic usage

```
int *p;
```

Tells compiler that variable p is the address of an `int`;

`*` called the “dereference operator” in this context;

```
p = &X;
```

Tells compiler to assign address of X to `p`;

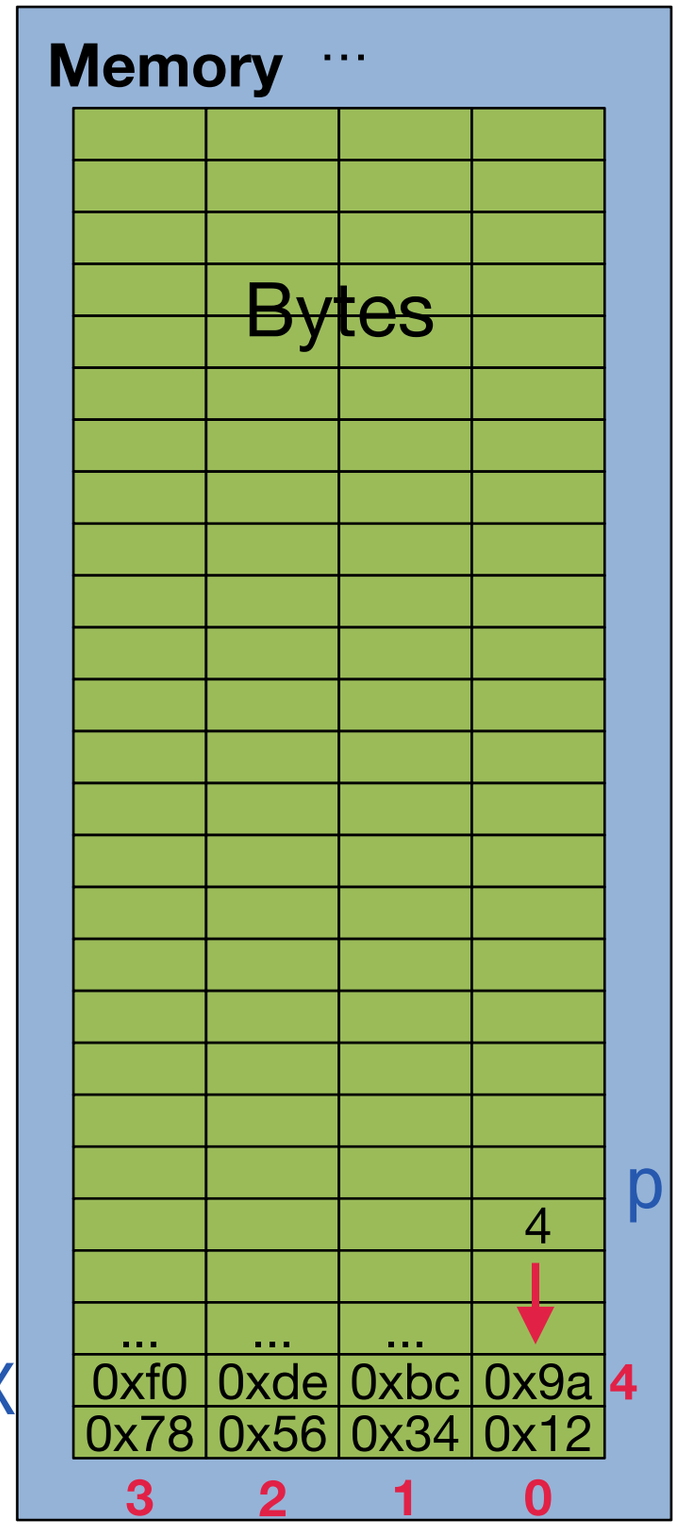
`&` called the “address operator” in this context;

```
z = *p;
```

Tells compiler to assign value at address p to `z`;

```
*p = 5;
```

Change the value at p to 5 and use pointer to write;



Pointers and Parameters 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

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

y equals to 4 now

Pointers and Structures

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

Point p1={1,2};
Point p2;
Point *paddr;
paddr = &p1;
```

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

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

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

Types of Pointers

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

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
  - 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
  - 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

Pointers in C

- 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 later
    - Dangling references and memory leaks

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 (a.k.a. “garbage”)
- What does the following code do?

Undefined Behavior

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

`ptr` not initialized.

Not even working in some environments

Outline

- Pointer
- Array
- Pointer arithmetic
- C memory management
  - Stack
  - Pointer

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

- Must specify size (or provide info. that can infer the size)

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'** (**NULL**) byte (a.k.a. "null terminator") (RTFM)

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

```
C89/90 & C99 standard: The declaration

        char s[] = "abc", t[3] = "abc";

defines ``plain'' char array objects s and t
whose members are initialized with character
string literals. This declaration is identical to

        char s[] = { 'a', 'b', 'c', '\0' },
             t[] = { 'a', 'b', 'c' };
```

Array/Pointer Duality

- Key Concept: Array variable is a “pointer” to the first (lowest addressed, i.e., 0th) element
- So, array variables almost identical to pointers
  - Not usually the other way (example next slide)
- 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

Array/Pointer Duality

- Be really careful!

```
char string1[] = "abc";
char *string2 = "abc";
```

- **CAN** modify `string1[*]`
- **CANNOT** modify `string2[*]`
- **CAN** access `string2` by `string2[*]`

Excercise

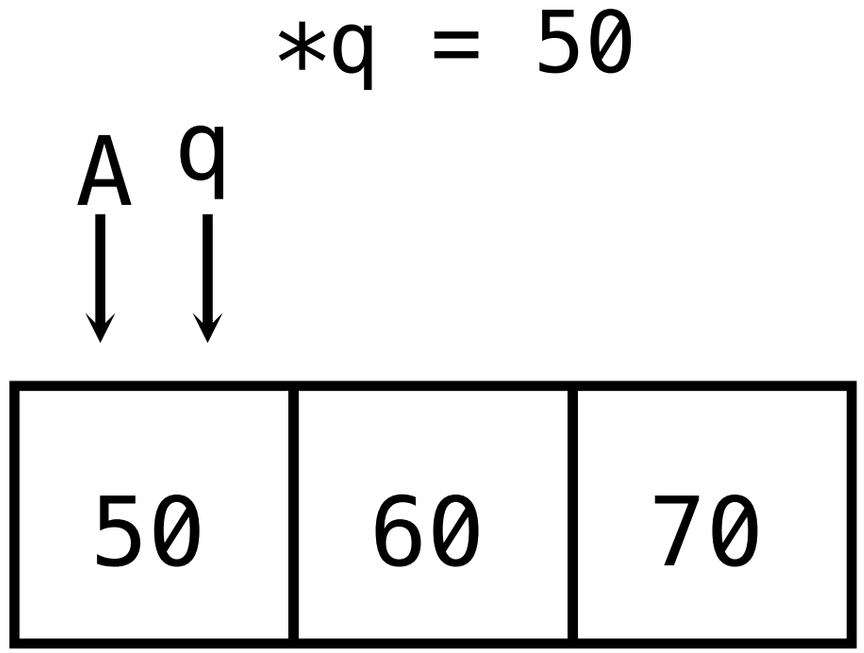
- What do these below mean? Be really careful!

```
int arr[] = { 3, 5, 6, 7, 9 };
int *p = arr;
int (*p1)[5] = &arr;
int *p2[5];
int (*p)(void);
int (*func_arr[5])(float x);
```

Changing a Pointer Argument?

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

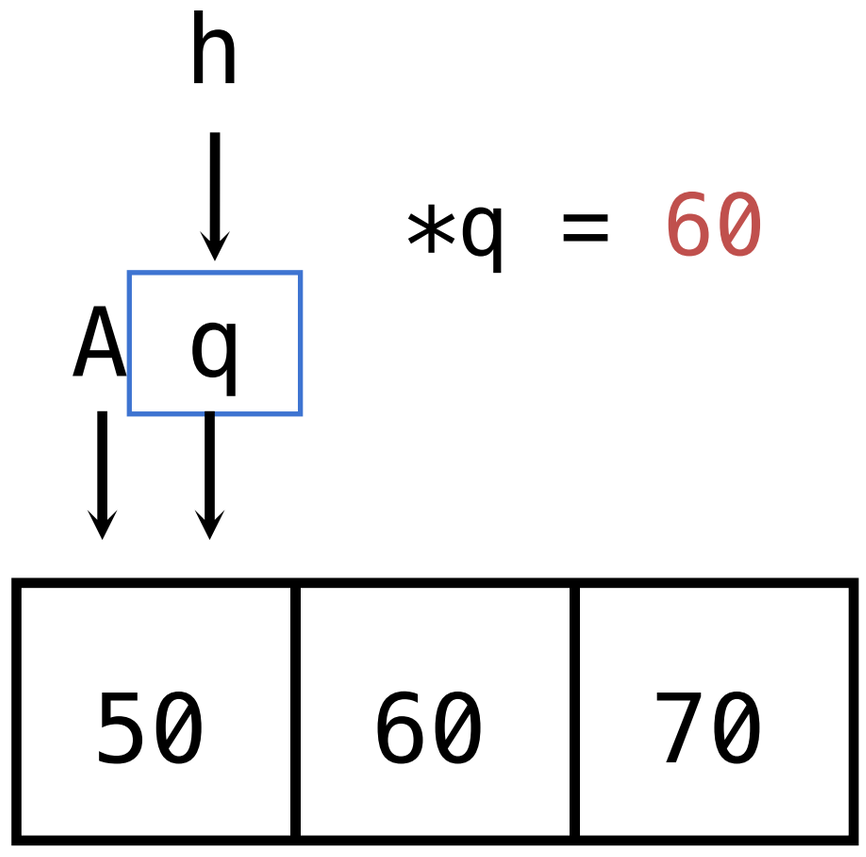
```
void inc_ptr(int *p)
{     p =  p + 1;    }

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



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);
```



Arguments in `main()`

- To get arguments to the main function, use:

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

  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 5:

    ```
    % clang  -ansi  introC_1_1.c  -o  introC_1_1.out
    ```

  - `argv` is a pointer array, pointing to multiple strings

Example

```
% clang  -ansi  introC_1_1.c  -o  introC_1_1.out
```

- argc = 5 /* number arguments */

  argv[0] = "clang",
  argv[1] = "-ansi",
  argv[2] = "introC_1_1.c",
  argv[3] = "-o",
  argv[4] = "introC_1_1.out"

More C Pointer & Array Dangers

- 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
  - Suggestion: We must pass the array and its size to any procedure that is going to manipulate it
- Out of boundary 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 and 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 avoid maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”

Arrays and Pointers

- Array = pointer to the initial (0^{th}) 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

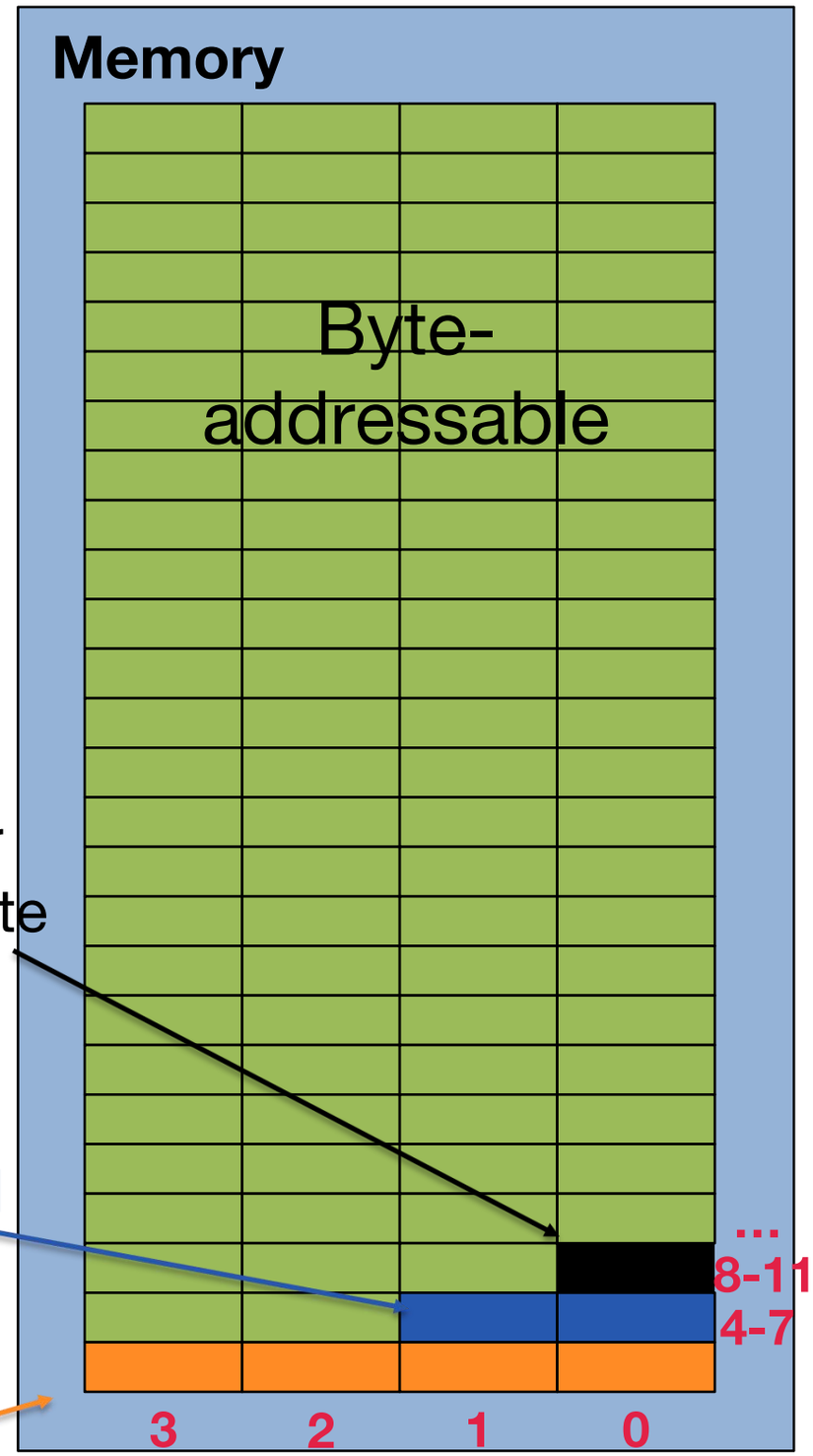
```
int foo(int array[],
    unsigned int size)
{
   … array[size – 1] …
}

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

Outline

- Pointer
- Array
- Pointer arithmetic
- C memory management
- Stack
- Pointer

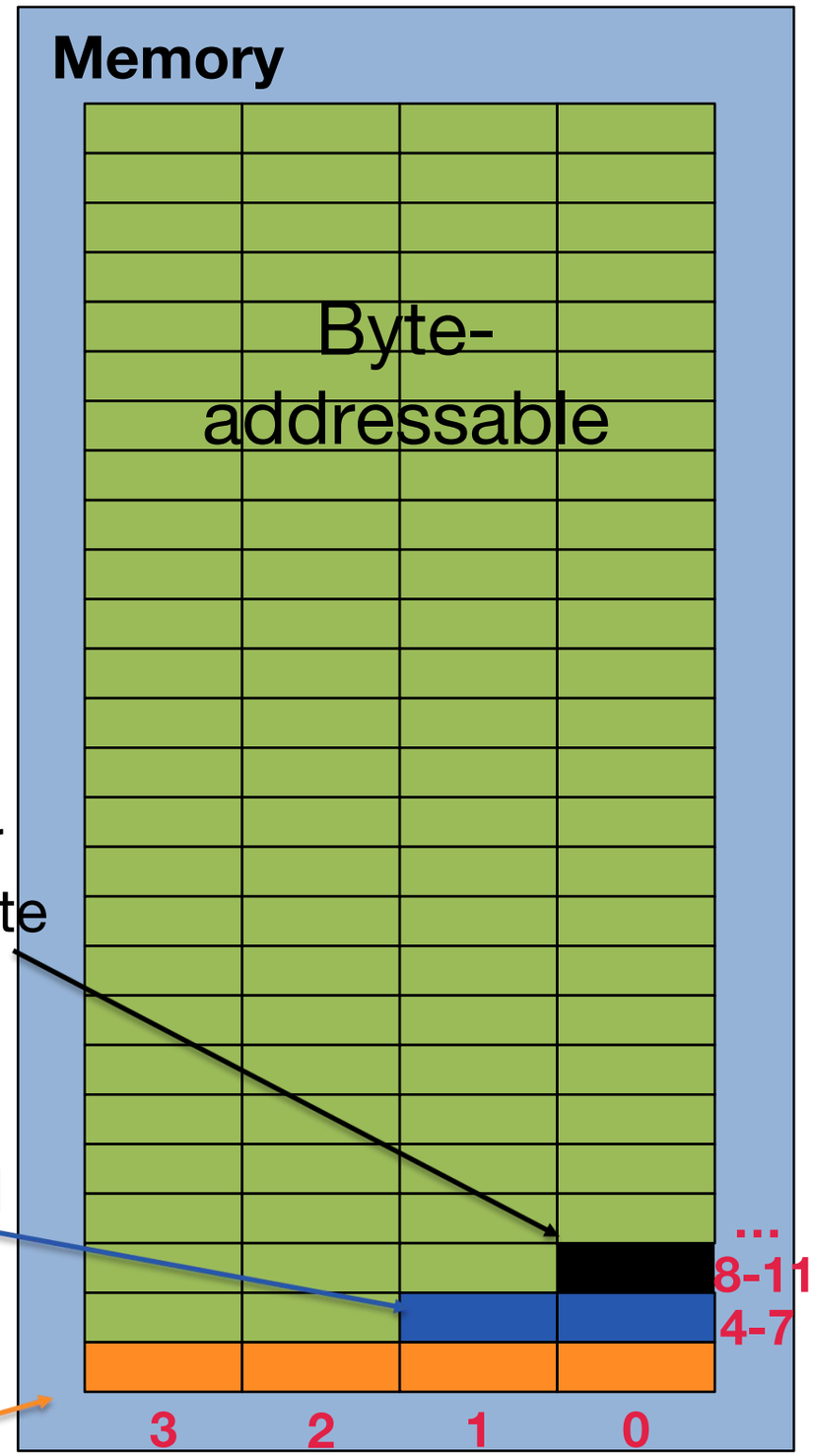
Pointing to Different-Sized Objects

- 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
- Alignment



Pointing to Different-Sized 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-bi
- Alignment




```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 1234, i32* %2, align 4
store i32 4321, i32* %3, align 4
%5 = load i32, i32* %2, align 4
%6 = load i32, i32* %3, align 4
%7 = add nsw i32 %5, %6
store i32 %7, i32* %4, align 4
```

`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(variable)`, or `sizeof(type)`
- We'll see more of `sizeof` when we look at dynamic memory management

Excercise

```
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 (32-bit address)?

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 (32-bit int)?

40

Pointer Arithmetic

pointer + *number* *pointer* – *number*

e.g., *pointer* + 1 adds 1 to a pointer

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

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

In each, p now points to b

(Assuming compiler doesn't reorder variables in memory. Never code like this!!!!)

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

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

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

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

Pointer arithmetic should be used cautiously

Pointer Arithmetic--Exercise

```
int arr[] = { 3, 5, 6, 7, 9 };
int *p = arr;
int (*p1)[5] = &arr;
```

Are arr+1, p+1, p1+1 the same?

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!

Concise `strlen()`

```
int(long) 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 last member of 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++;
```

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

Valid Pointer Arithmetic

- Add an integer to a pointer.
- Subtract 2 pointers (pointed to the same array/object)
- 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

Summary

- “Lowest High-level language”
  - => closest to assembler

- Pointers: powerful but dangerous

- Pointer arithmetic and arrays useful but also dangerous

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.

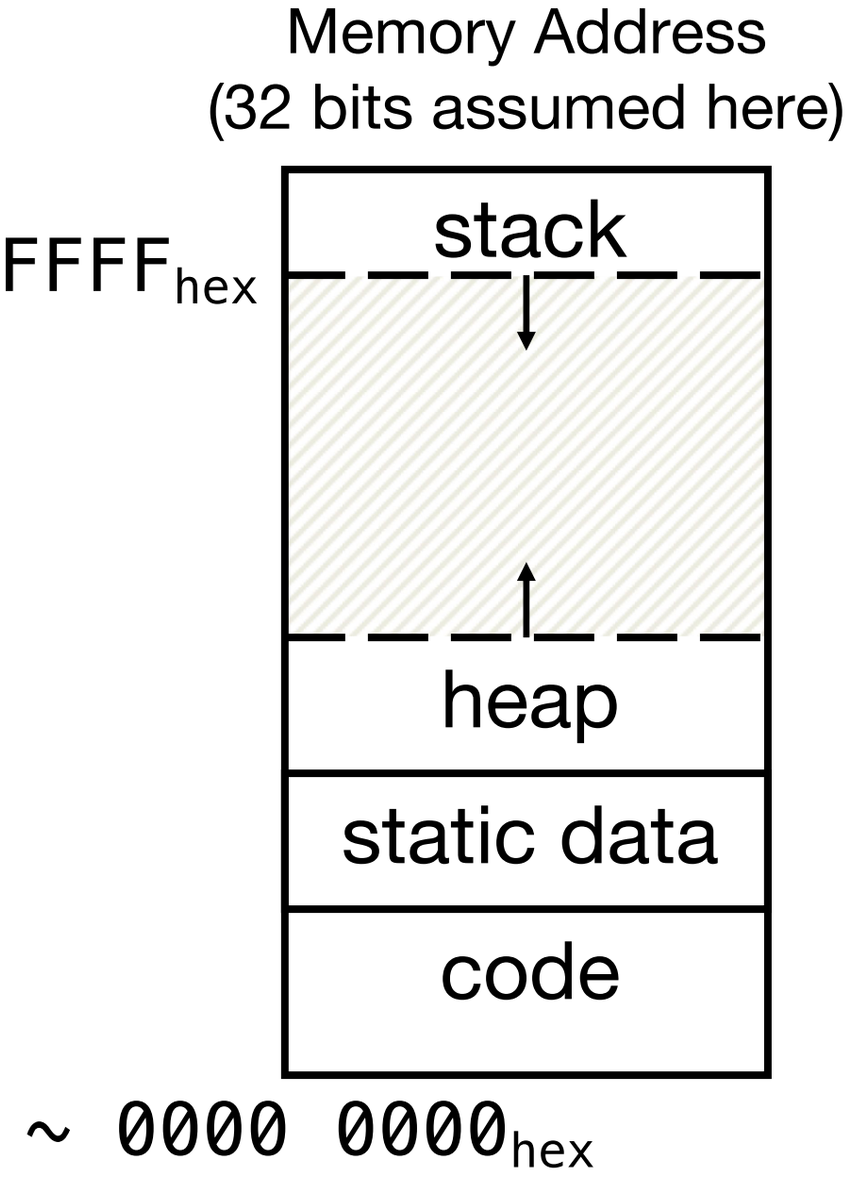
"C gives you a lot of extra rope but be careful not to hang yourself with it!"

Outline

- Pointer
- Array
- Pointer arithmetic
- C memory management
  - Stack
  - Pointer

C Memory Management

- To simplify, assume one program runs at a time
- A 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 (a.k.a. text): loaded when program starts, does not change
  - 0x0 unwritable/unreadable (NULL pointer)



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

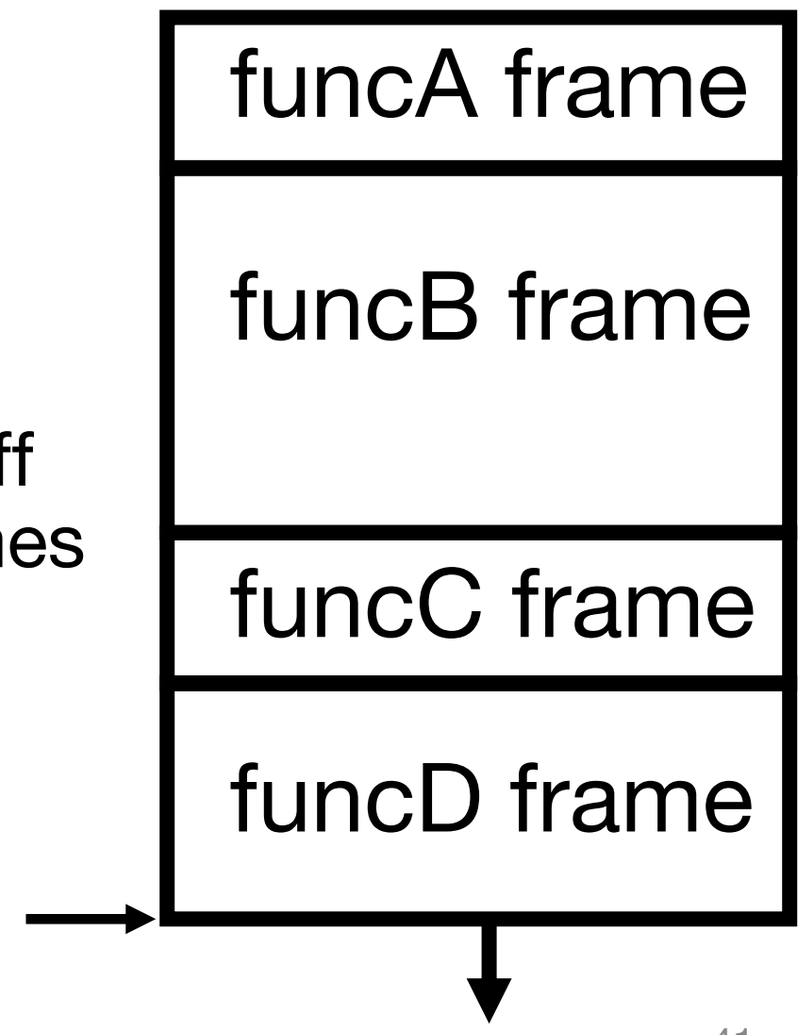
```
int myGlobal;
main() {
   int myTemp;
}
```

- For the above two types, the memory management is automatic
  - Don’t need to deallocating when no longer using them
  - A variable does not exist anymore once a function ends!

The Stack

- Every time a function is called, a new “stack 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
- Details covered later (RISC-V processor)

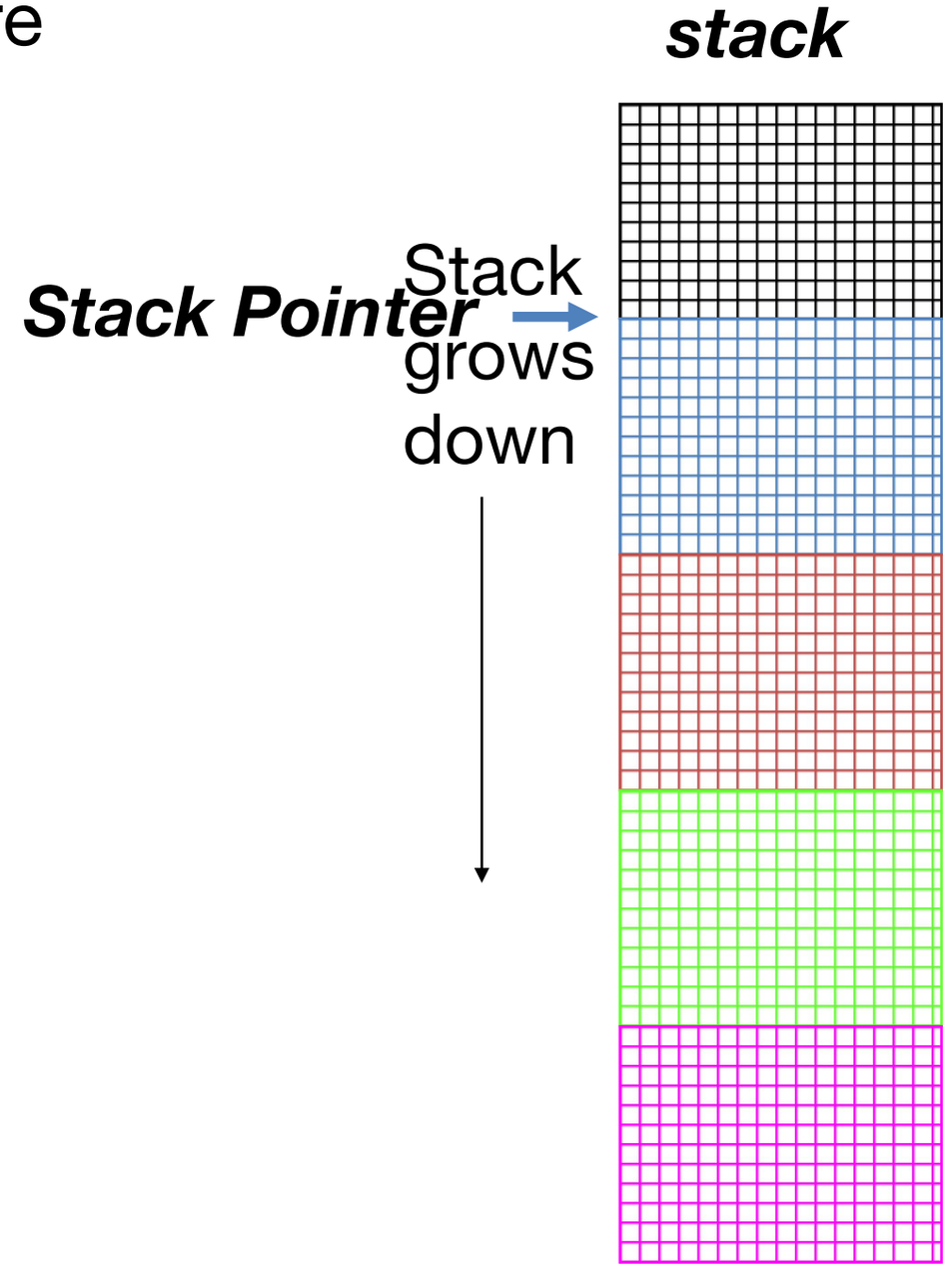
```
funcA() { funcB(); }
funcB() { funcC(); }
funcC() { funcD(); }
```



Stack Animation

- Last In, First Out (LIFO) data structure

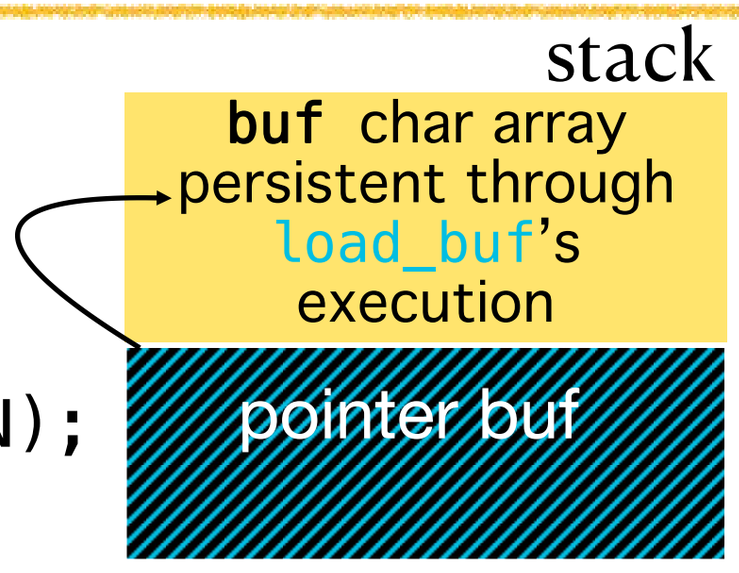
```
main ()
{ a(0);
}
   void a (int m)
   { b(1);
   }
     void b (int n)
     { c(2);
     }
       void c (int o)
       { d(3);
       }
        void d (int p)
        {
        }
```



Passing Pointers into the Stack

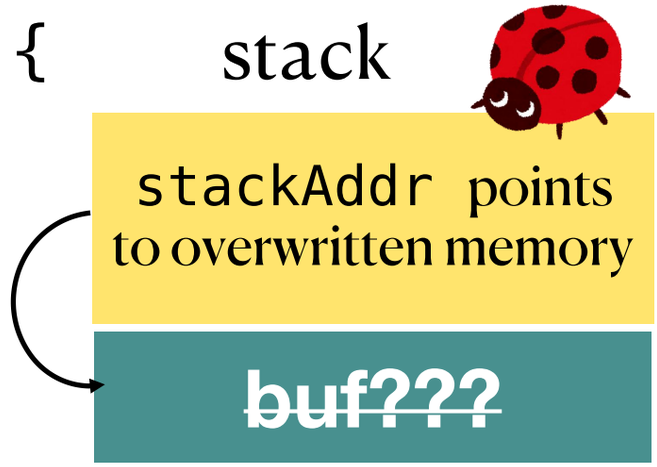
- It is fine to pass a pointer to stack space further down.

```
#define BUFLEN 256
int main() {
    …
    char buf[BUFLEN];
    load_buf(buf, BUFLEN);
    …
}
```



- However, it is bad to return a pointer to something in the stack!
  - Memory will be overwritten when other functions called!
  - So your data would no longer exist, and writes can overwrite key pointers, causing crashes!

```
char* make_buf() {
    char buf[50];
    return buf;
}

int main(){
    …
    char *stackAddr =
        make_buf();
    …
}
```



Carving on the moving boat to look for the sword

Managing the Heap

- The heap is dynamic memory – memory that can be allocated, resized, and freed during program runtime.
  - Useful for persistent memory across function calls
  - But biggest source of pointer bugs, memory leaks, …
- Large pool of memory, not allocated in contiguous order
  - Back-to-back requests for heap memory could result in blocks very far apart
- C supports four 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 (might move)

Managing the Heap

- **`void *malloc(size_t n):`**
  - Allocate a block of uninitialized memory
  - **`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; always check for return **`NULL (if (ip))`**
  - Think of pointer as a handle that describes the allocated block of memory; Additional control information stored in the heap around the allocated block! (Including size, etc.)

- Examples:

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

```
int *ip1, *ip2;
ip1 = (int *) malloc(sizeof(int));
Ip2 = (int *) malloc(20*sizeof(int)); //allocate an array of 20 ints.

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

Assuming size of objects can lead to misleading, unportable code. Use **`sizeof()`**!

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

Managing the Heap

- **`void *realloc(void *p, size_t size)`:**
  - Returns new address of the memory block.
    - In doing so, it may need to copy all data to a new location.

```
realloc(NULL, size); // behaves like malloc
realloc(ptr,  0); // behaves like free, deallocates heap block
```

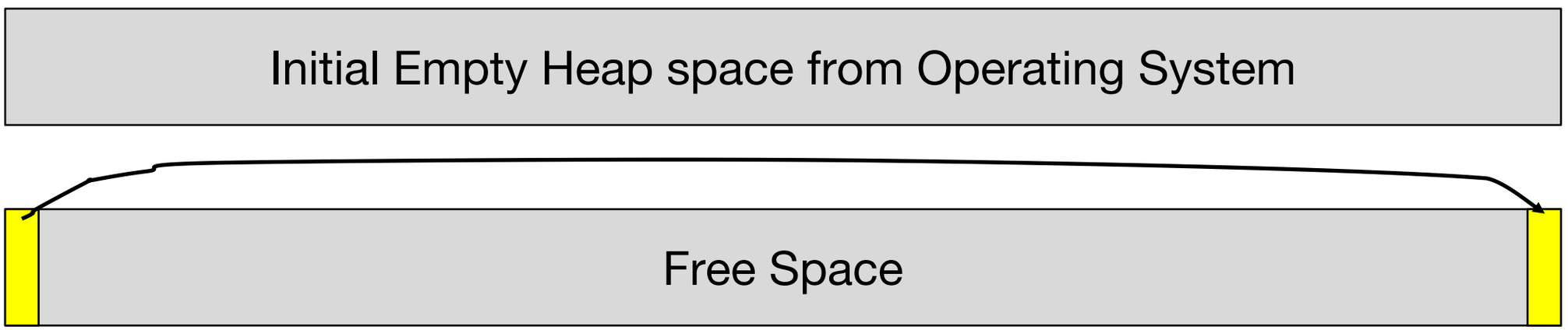
  - Always check for return NULL

Keep track of this, since it might change.

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

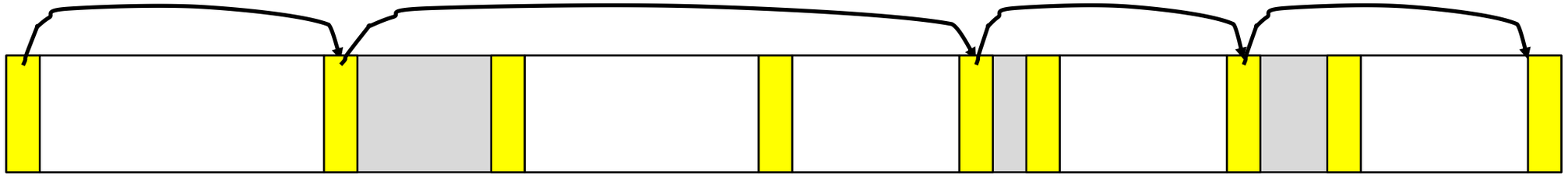
How are `malloc/free` implemented?

- Underlying operating system allows `malloc` library to ask for large blocks of memory to use in heap (`stdlib.h`)



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

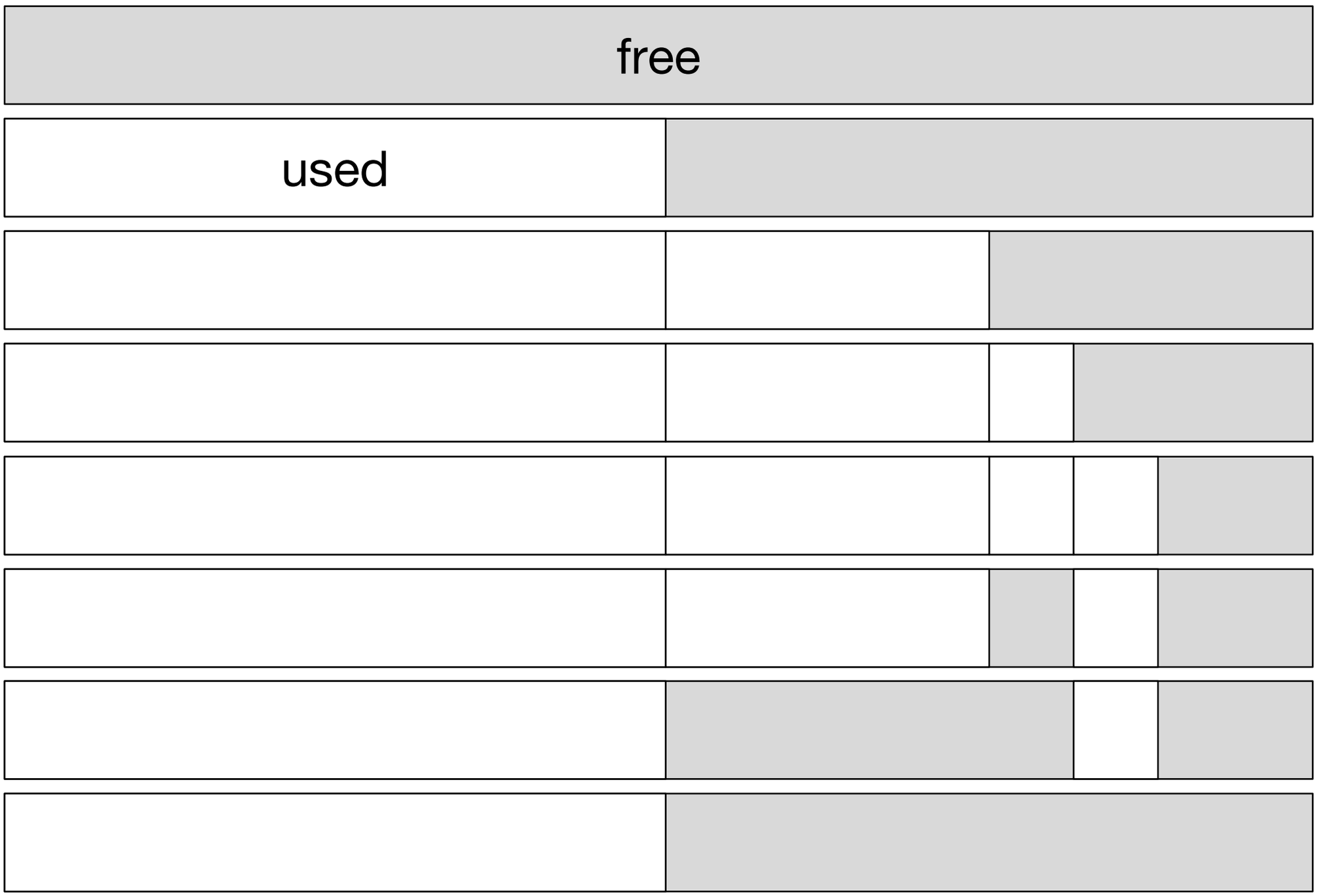
First allocation chews up space from start of free space



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

- "Buddy allocators" always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks



Observations/Summary

- 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, avoid “dangling references"
- Managing the heap is tricky:
  - Memory can be allocated/deallocated at any time
  - “Memory leak”: If you forget to deallocate memory
  - “Use after free”: If you use data after calling free
  - “Double free”: If you call free 2x on same memory