

Memory Allocation (Start)

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Sid Sannapareddy Maya Huizar

pollev.com/tqm

❖ Any Questions about PennOS?

Administrivia

- ❖ Final Exam; July 31st
 - Same format as the Midterm!
 - Last course day is July 23rd
 - Final Exam Review Thursday, July 24th
- ❖ PennOS Due Friday, July 25th
- ❖ Final Grades due August 11th
 - So even though the course is “over”, there’s still wiggle room at the end.

Administrivia

❖ Some notes:

- Reminder, you instead of just doing:

you may need to do:

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

```
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);  
lseek(FAT_FD, offset, SEEK_SET);  
write(FAT_FD, contents, size);
```

- With the description of `setitimer()`, it just says that sigalarm is delivered to the process, not necessarily the calling thread. To make sure sigalarm goes to the scheduler, you may want to make it so that all threads (spthread or otherwise) that aren't the scheduler call something like: **`pthread_sigmask(SIG_BLOCK, SIGALARM)`**
 - Which will block SIGALARM in that thread.

Administrivia

- ❖ If you are having issues with the scheduler not running you can try running
 - `strace -e 'trace=!all' ./bin/pennos`
 - You may have to install strace: `sudo apt install strace`
 - This will print out every time a signal is sent to your pennos
 - (Usual fix is the `pthread_sigmask` thing on the previous slide)

Lecture Outline

- ❖ **Stack & Heap w/ Free-lists**
- ❖ Memory Alignment
- ❖ Fragmentation
- ❖ Leaks

Stack & Heap

- ❖ Hopefully you are familiar with the stack and the heap,
 - Quick refresher now though
- ❖ Stack:
 - Where local variables & information for local functions are stored (return address, etc).
 - Grows whenever you call a function. pushes a “stack frame” for each function call.
- ❖ Heap:
 - Dynamically allocated data stored here. Usually done when data needs to exist beyond the scope it is allocated in, or the size is not known at compile time

Stack Example:

Stack frame for main is
created when CPU
starts executing it

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    → int sum = sum(3);
    printf("sum: %d\n", sum);
    return EXIT_SUCCESS;
}
```

int sum;

Stack frame for
main()

Stack Example:

```
#include <stdio.h>
#include <stdlib.h>

→ int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    printf("sum: %d\n", sum);
    return EXIT_SUCCESS;
}
```

```
int sum;
```

Stack frame for
main()

```
int i;
int sum;
int n;
```

Stack frame for
sum()

Stack Example 1:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    printf("sum: %d\n", sum);
    return EXIT_SUCCESS;
}
```

int sum;

Stack frame for
main()

sum()'s stack frame
goes away after
sum() returns.

main()'s stack frame
is now top of the stack
and we keep executing
main()

Stack Example:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int sum = sum(3);
    printf("sum: %d\n", sum);
    return EXIT_SUCCESS;
}
```

int sum;

Stack frame for
main()

????

Stack frame for
printf()

Stack

- ❖ Grows, but has a static max size
 - Can find the default size limit with the command `ulimit -all`
(May be a different command in different shells and/or linux versions. Works in bash on Ubuntu though)
 - Can also be found at runtime with `getrlimit(3)`

- ❖ Max Size of a stack can be changed
 - at run time with `setrlimit(3)`
 - At compilation time for some systems (not on Linux it seems)
 - (or at the creation of a thread, more on threads next lecture)

The Heap

- ❖ The Heap is a large pool of available memory to use for Dynamic allocation
- ❖ This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- ❖ **malloc:**
 - searches for a large enough unused block of memory
 - marks the memory as allocated.
 - Returns a pointer to the beginning of that memory
- ❖ **free:**
 - Takes in a pointer to a previously allocated address
 - Marks the memory as free to use.

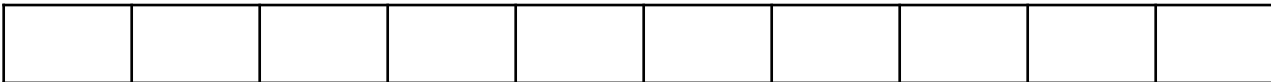
Dynamic Memory Example (Naïve)

- ❖ When we allocate data on the heap we get the guarantee that the data is contiguous within an allocation

```
#include <stdlib.h>

int main() {
    → char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

- ❖ Heap:



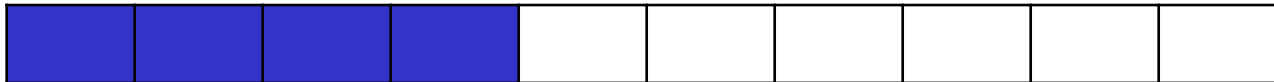
Dynamic Memory Example (Naïve)

- ❖ When we allocate data on the heap we get the guarantee that the data is contiguous within an allocation

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    → char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

- ❖ Heap:



Dynamic Memory Example (Naïve)

- ❖ When we allocate data on the heap we get the guarantee that the data is contiguous within an allocation

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

- ❖ Heap:



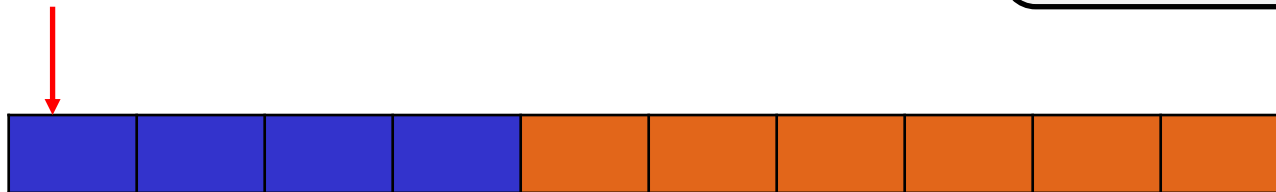
Dynamic Memory Example (Naïve)

- ❖ When we allocate data on the heap we get the guarantee that the data is contiguous within an allocation

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

- ❖ Heap:



How do we know how much to deallocate?

How do we mark the memory as “free”?

When we allocate next, how do we know where we could allocate from?

Free Lists

- ❖ One way that malloc can be implemented is by maintaining an implicit list of the space available and space allocated.
- ❖ Before each chunk of allocated/free memory, we'll also have this metadata:

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```

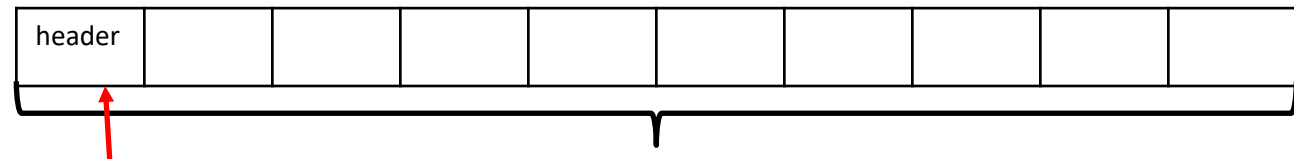
Dynamic Memory Example

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

This diagram is
not to scale

❖ free_list ->



```
{
    NULL,
    NULL,
    false,
    1000
}
```

The metadata is at
the beginning of the
chunk of memory

Dynamic Memory Example

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

Free chunks can
be split to
allocate blocks of
specific size

Malloc gets a
pointer to just
after the
metadata

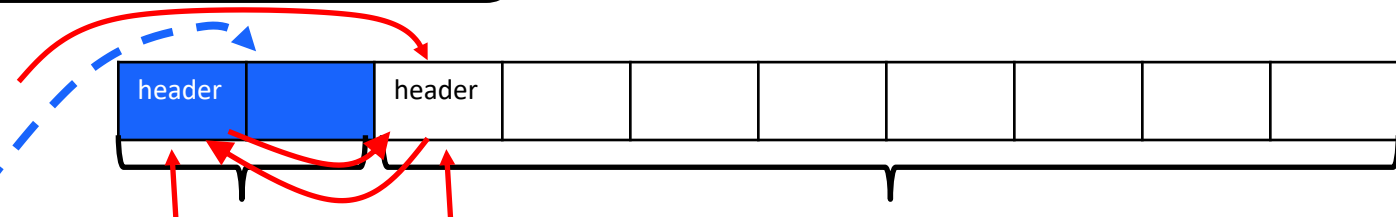
❖ free_list

malloc
return
value

```
{
    NULL,
    0x...,
    true,
    4
}
```

```
{
    0x...,
    NULL,
    false,
    972
}
```

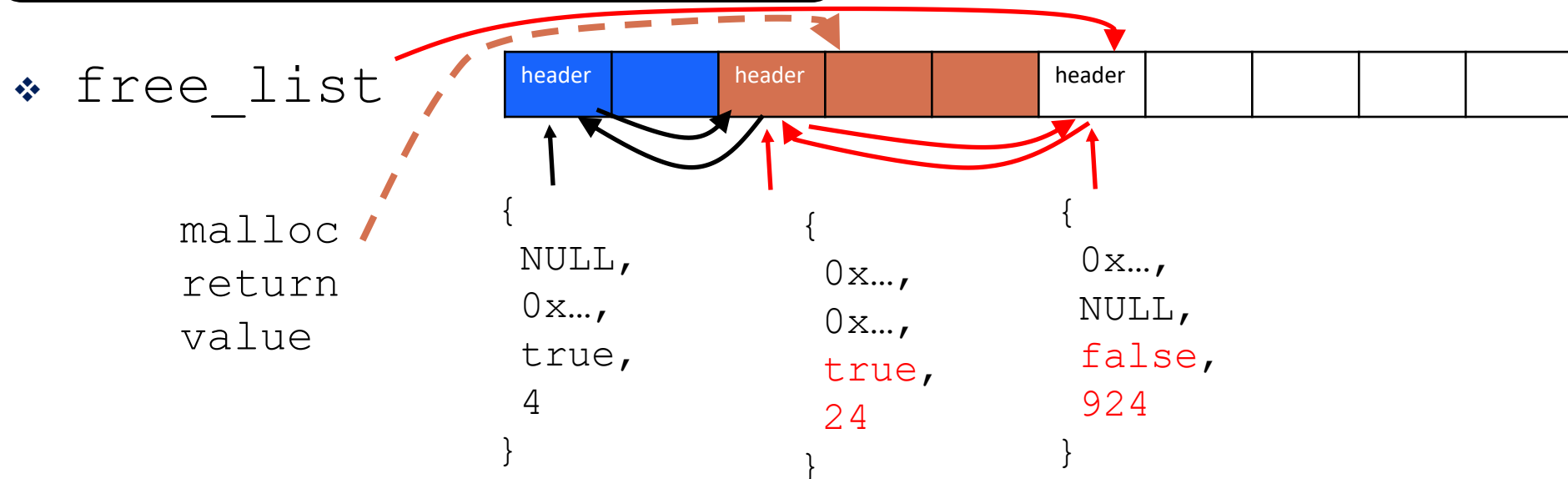
free_list
points to first
free chunk



Dynamic Memory Example

```
#include <stdlib.h>

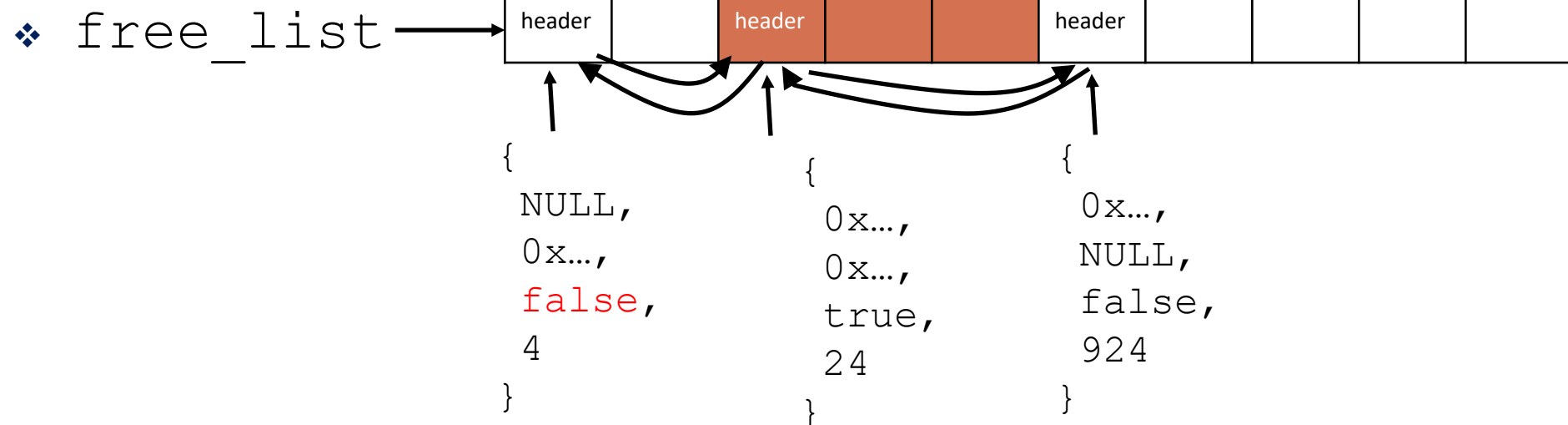
int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```



Dynamic Memory Example

```
#include <stdlib.h>

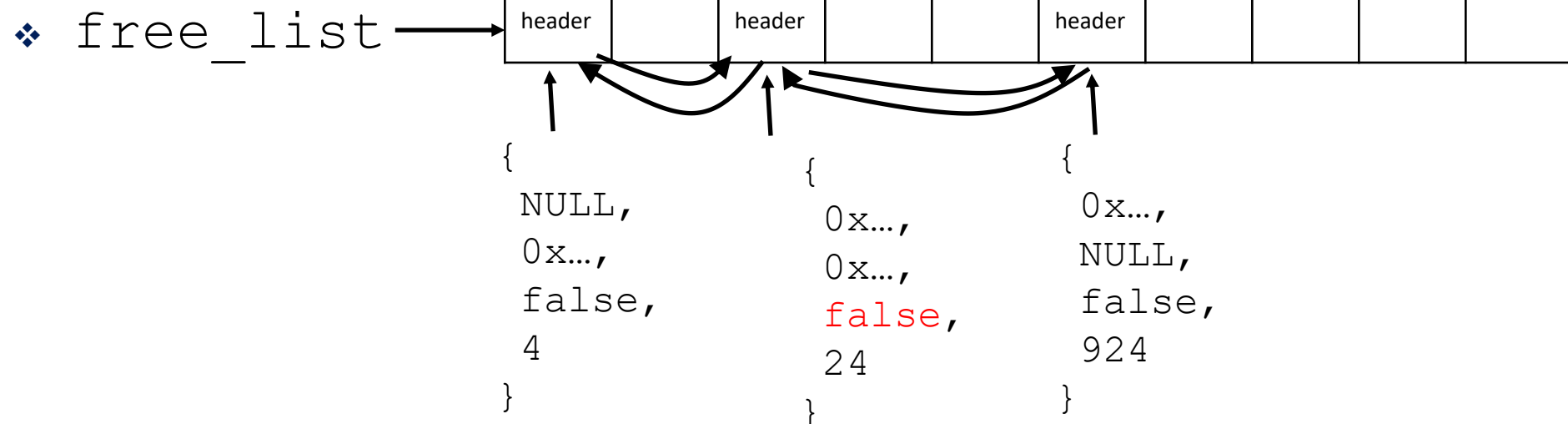
int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    → free(ptr);
    free(ptr2);
}
```



Dynamic Memory Example

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    → free(ptr2);
}
```



Dynamic Memory Example

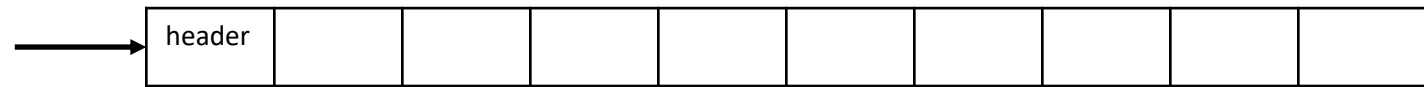
```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    free(ptr2);
}
```

Once a block has been freed, we can try to "coalesce" it with their neighbors

The first free couldn't be coalesced, only neighbor was allocated

❖ free_list



```
{
    NULL,
    0x...,
    false,
    1000
}
```


Heap

- ❖ **malloc()** and **free()** are not system calls, they are implemented as part of the C std library
 - **malloc()** and **free()** will sometimes internally invoke system calls to expand the heap if needed
 - Instead, these functions just manipulate memory already given to the process, marking some as free and some as allocated
- ❖ System calls used by **malloc()** and **free()**:
 - **brk()** and **sbrk()**
 - Used to grow/shrink the data segment of memory
 - **mmap()**, **munmap()**
 - creates / or destroys a mapping in virtual address space

Memory Allocation Has a Cost

- ❖ There is a reason we had “Unnecessary Memory Allocation” in the style guide.
- ❖ Memory Allocation is not an $O(1)$ operation

- ❖ It takes time to:
 - Search for a block size that is big enough
 - Coalesce / free memory
 - Grow the heap if needed
 - In multithreaded applications, locks need to be acquired!

pollev.com/tqm

❖ How many memory allocations occur in each piece of code?

- Assume vector resizes will double capacity
- `std::vector` is an arraylist in C++
`std::list` is a linked list in C++

```
int main() {  
    vector nums {4, 8}; // size and capacity == 2  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

```
int main() {  
    list nums {4, 8};  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

pollev.com/tqm

❖ How many memory allocations occur in each piece of code?

- Assume vector resizes will double capacity
- `std::vector` is an arraylist in C++
`std::list` is a linked list in C++

```
int main() {  
    vector nums {4, 8}; // size and capacity == 2  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

3

```
int main() {  
    list nums {4, 8};  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
}
```

6

Minimizing Allocations

- ❖ As we saw previously, memory allocations require time, sometimes a lot of time to compute.
- ❖ If performance is our goal, we should minimize the number of allocations we make.
- ❖ This can include
 - Making references instead of copies
 - Using functions like `vector::reserve(size_t new_capacity)`
 - In C++
 - Java arraylist lets you specify capacity in the constructor.
 - Using move semantics

Lecture Outline

- ❖ Stack & Heap w/ Free-lists
- ❖ **Memory Alignment**
- ❖ Fragmentation
- ❖ Leaks

pollev.com/tqm

- ❖ What do you think `sizeof(alloc_info)` is on our 64-bit machines? (how many bytes is it)
- ❖ Assume `size_t` is 4 bytes.

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```

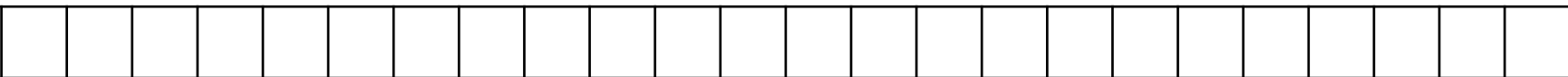
Memory Alignment

- ❖ In memory, data isn't always just crammed together.
- ❖ Hardware likes it so that if we are dealing with a 4-byte type, then that variable is stored at an address that is a multiple of 4 bytes.
 - Same with types that are 8, 2, 1-byte etc.
- ❖ This isn't always the case, but our software and hardware tries to make this the case.

Back to Poll:

❖ How big is this struct?

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```



Back to Poll: (Naïve answer)

❖ How big is this struct?

- `prev`
- `next`
- `allocated`
- `size`

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```



Back to Poll: (Fragmentation answer)

❖ How big is this struct?

- `prev`
- `next`
- `allocated`
- `size`

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```



Fragmentation: Struct Size

- ❖ C structs will also try to be a multiple of its biggest member.

So in our example, we want to make sure that the struct size is a multiple of 8

```
// this is simplified  
// not what malloc really does  
struct alloc_info {  
    alloc_info* prev;  
    alloc_info* next;  
    bool allocated;  
    size_t size;  
};
```

- ❖ Given this struct foo: what is the size of the struct?
What is the optimal size of the struct we could have
if we rearranged the fields and still respected alignment?

```
struct foo {  
    bool allocated;  
    uint32_t size;  
    bool flag;  
    uint16_t bleg;  
};
```

Lecture Outline

- ❖ Stack & Heap w/ Free-lists
- ❖ Memory Alignment
- ❖ **Fragmentation**
- ❖ Leaks

Fragmentation

- ❖ Fragmentation: when storage is used inefficiently, which can hurt performance and ability to allocate things.

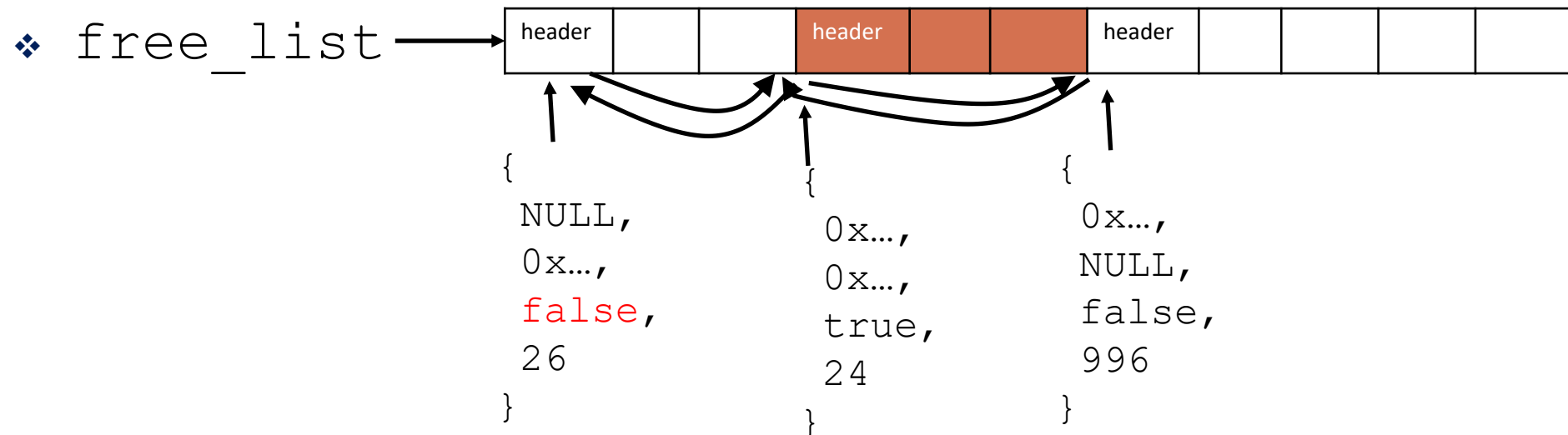
Specifically, when there is something that prevents "unused" memory from otherwise being used

- ❖ External Fragmentation: when free memory is spread out over small portions that cannot be coalesced into a bigger block that can be used for allocation

External Fragmentation Example

```
#include <stdlib.h>

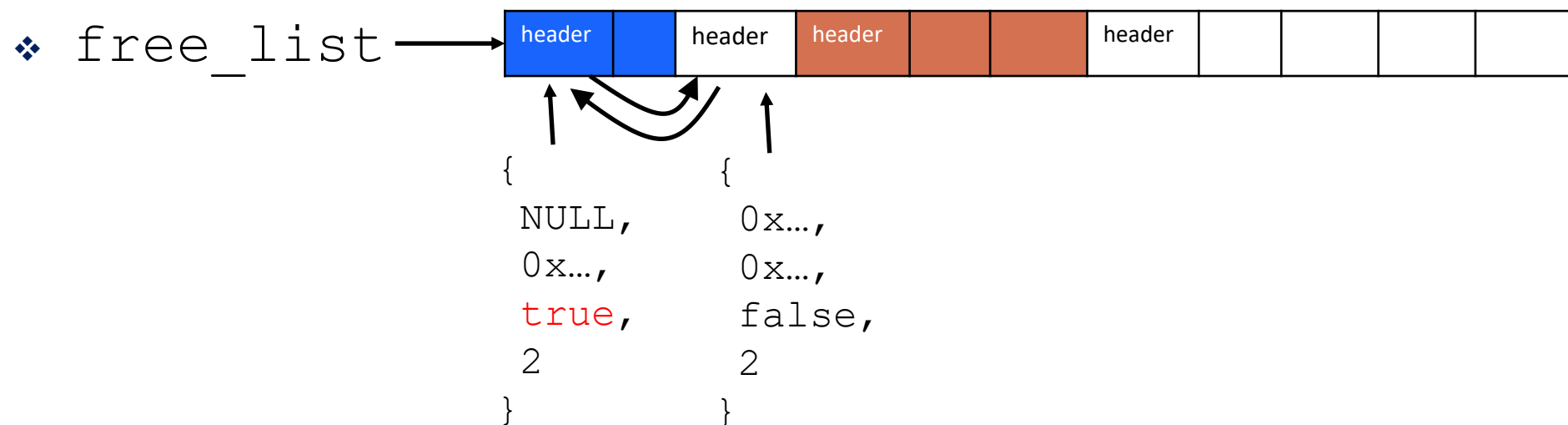
int main() {
    char* ptr = malloc(26*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    → free(ptr);
    ptr = malloc(2*sizeof(char));
    ...
}
```



External Fragmentation Example

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    ptr = malloc(2*sizeof(char));
    ...
}
```



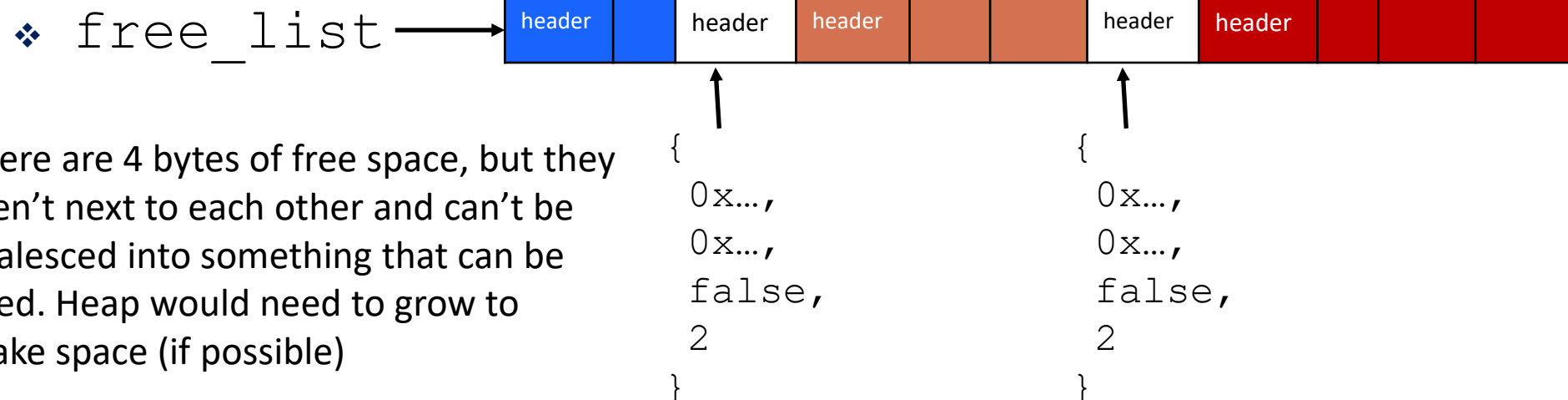
External Fragmentation Example

```
#include <stdlib.h>

int main() {
    char* ptr = malloc(4*sizeof(char));
    char* ptr2 = malloc(6*sizeof(int));
    ...           // do stuff with ptr
    free(ptr);
    ptr = malloc(2*sizeof(char));
    ...
}
```

After some more series of allocations and frees (not shown), we get this:

Let's say `malloc(4)` gets called (trying to allocate 4 bytes) what happens?



Internal Fragmentation

- ❖ Internal Fragmentation: When more space is allocated for something than is actually used. This fragmentation happens “internally” within an allocated portion, instead of “external” to one.
- ❖ What if someone calls `malloc(4096 * sizeof(char*))` and only uses the first `char*`?
 - Can be thought of internal fragmentation, not the allocator's fault though (in this use case)
- ❖ What if we allocate a struct that has empty space to meet alignment requirements?
- ❖ Sometimes we call `malloc()` and more space is allocated than needed.
 - if we allocate for 7 bytes, 8 may actually be allocated. Computer may want addresses to be aligned to a multiple of a power of 2

First Fit

- ❖ There may be multiple free blocks that can be chosen for allocation.
- ❖ The allocation policy we used in our examples is First Fit: find the first block of memory that is big enough
 - Start at the front of the free list, iterate till we find something big enough
 - Usually the simplest to implement

Best Fit

- ❖ **Best Fit**: another approach where instead you look for the portion of memory that is the “best” or “tightest” fit
- ❖ If allocating for 4 bytes of memory, search for the smallest block that is ≥ 4 bytes.

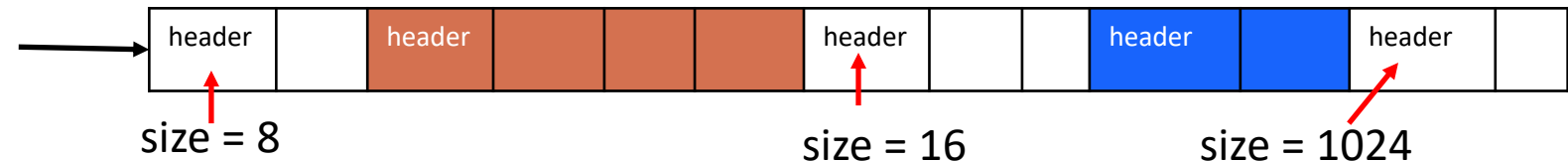
Worst Fit

- ❖ **Worst Fit**: another approach where instead you look for the portion of memory that is the “worst” fit (opposite of best fit)
- ❖ If allocating for 4 bytes of memory, search for the **largest** block that is ≥ 4 bytes.

- ❖ What is the approximate runtime of the algorithms? (e.g. $O(N \log(N))$). What is the best/worst case?
 - First Fit
 - Best Fit
 - Worst Fit

- ❖ Lets say we call **malloc (4 bytes)** . Which block is allocated in this example if we choose:
 - First Fit
 - Best fit
 - Worst fit

free_list

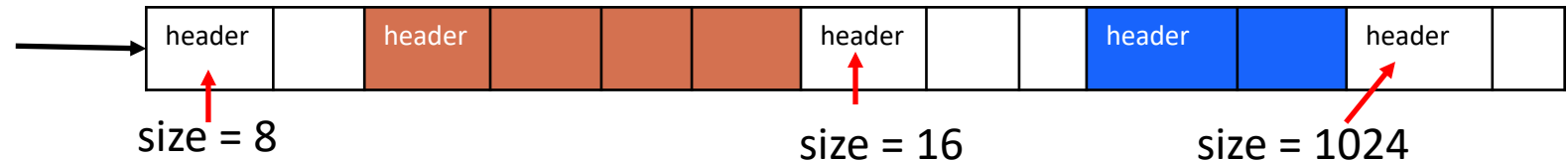




pollev.com/tqm

- ❖ What is the approximate runtime of the algorithms? (e.g. $O(N \log(N))$). What is the best/worst case?
 - First Fit **best: $O(1)$, worst: $O(n)$**
 - Best Fit **best: $O(n)$, worst: $O(n)$**
 - Worst Fit **best: $O(n)$, worst: $O(n)$**
- ❖ Lets say we call **malloc(4 bytes)** . Which block is allocated in this example if we choose:
 - First Fit **8 byte chunk**
 - Best fit **8 byte chunk**
 - Worst fit **1024 byte chunk**

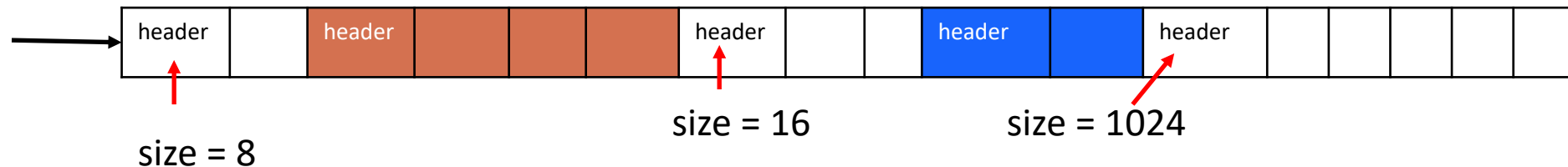
```
free_list
```



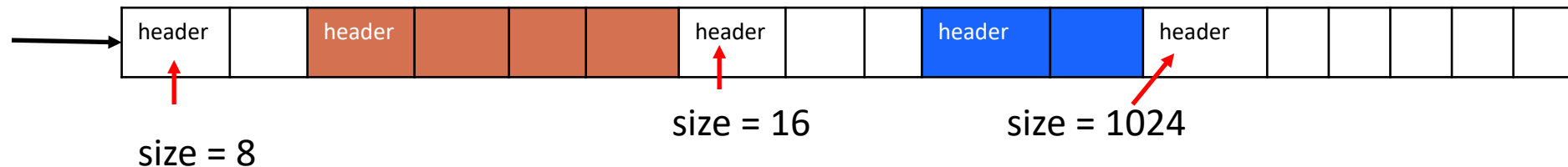
pollev.com/tqm

- ❖ It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?

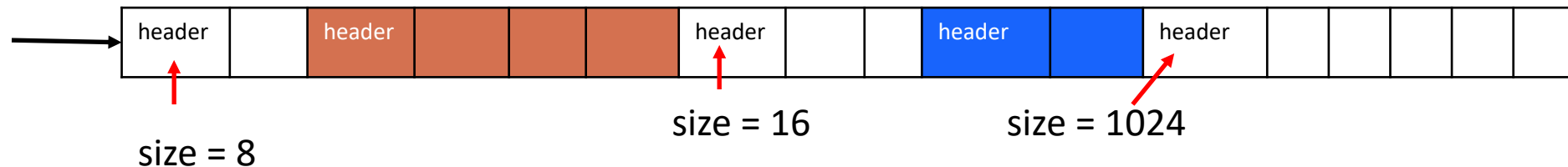
- ❖ It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?
- ❖ Less small “leftover” fragments, fragments are bigger and easier to reuse
- ❖ In the previous example, if we allocate for size 6...



- ❖ It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?
- ❖ Less small “leftover” fragments, fragments are bigger and easier to reuse
- ❖ In the previous example, if we allocate for size 6...
 - Best fit would allocate the size 8 free chunk leaving a size 2 chunk that is unlikely to be usable



- ❖ It turns out that over long periods of time, worst fit can work better than best fit. Why is this the case?
- ❖ Less small “leftover” fragments, fragments are bigger and easier to reuse
- ❖ In the previous example, if we allocate for size 6...
 - Worst fit would use 1024, splitting it into 6 and 1018. 8 chunk is still usable and 1018 is still usable.



Lecture Outline

- ❖ Stack & Heap w/ Free-lists
- ❖ Memory Alignment
- ❖ Fragmentation
- ❖ **Leaks**

Memory Leaks

- ❖ How do we feel about them? Good? Bad?

Memory Leaks

- ❖ The most common Memory Pitfall
- ❖ What happens if we allocate something, but don't delete it?
 - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **free**-d
 - Is this a problem?

Memory Leaks

- ❖ The most common Memory Pitfall
- ❖ What happens if we allocate something, but don't delete it?
 - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **free**-d
 - Is this a problem?
- ❖ If this happens enough, we run out of heap space and program may slow down and eventually crash

Memory Leaks

- ❖ The most common Memory Pitfall
- ❖ What happens if we allocate something, but don't delete it?
 - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **free**-d
 - Is this a problem?
- ❖ What if it is a short lived program or we are about to exit the process? Do we still need to free?
 - Eh..... The OS will clean up all of memory when our process exits
 - What if we were about to exit a thread?
 - Probably still cleanup, threads share an address space.

Memory Leaks

- ❖ The most common Memory Pitfall
- ❖ What happens if we allocate something, but don't delete it?
 - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **free**-d
- ❖ Garbage Collection
 - Automatically “frees” anything once the program has lost all references to it
 - Affects performance, but avoid memory leaks
 - Java and other “high level” languages
- ❖ RAI (Resource Acquisition Is Initialization)
 - C++ and Rust have this, it is VERY GOOD

RAII

- ❖ In C++, Rust and other languages we have RAII
 - Resource Acquisition is Initialization
 - What this really means is that in addition to a “constructor” for an object there exists a “destructor” that cleans up the object
 - The destructor is called for you when the object falls out of scope
The destructor will free the underlying memory the vector allocated!
 - Can still cause issues, but makes it easier than C’s explicit calls to free()

```
int main() {  
    vector nums {4, 8};  
    nums.push_back(5);  
    nums.push_back(9);  
    nums.push_back(5);  
    nums.push_back(0);  
    // nums.~vector()    implicit destructor call  
}
```

```
int main() {  
    if (...) {  
        vector nums {4, 8};  
        nums.push_back(0);  
        // nums.~vector()  
    }  
}
```

Safety C Example

- ❖ Here is an example in C where is the issue?

```
int main(int argc, char** argv) {  
    int* ptr = malloc(sizeof(int));  
    assert(ptr != NULL);  
    *ptr = 5;  
  
    // do stuff with ptr  
  
    free(ptr);  
  
    printf("%d\n", *ptr);  
}
```

C++ Safety

- ❖ Here is an example in C++ where is the issue?

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char** argv) {
    vector<int> v {3, 4, 5};
    int* first = &v.front();

    cout << *first << endl;    // print(*first)

    v.push_back(6);

    cout << v.size() << endl;  // print(v.size())
    cout << *first << endl;    // print(*first)
}
```

C++ Safety

- ❖ Here is an example in C++ where is the issue?

```
#include <iostream>
#include <vector>

using namespace std;

int* foo() {
    vector<int> v {3, 4, 5};
    return &v[0];
}

int main(int argc, char** argv) {
    int* first = foo();
    cout << *first << endl;    // print(*first)
}
```

More Next Time 😊

- ❖ Next Time
 - Garbage Collection
 - Arena Allocators
 - Slab Allocators
 - Buddy Allocators