Dynamic Memory Allocation: Introduction

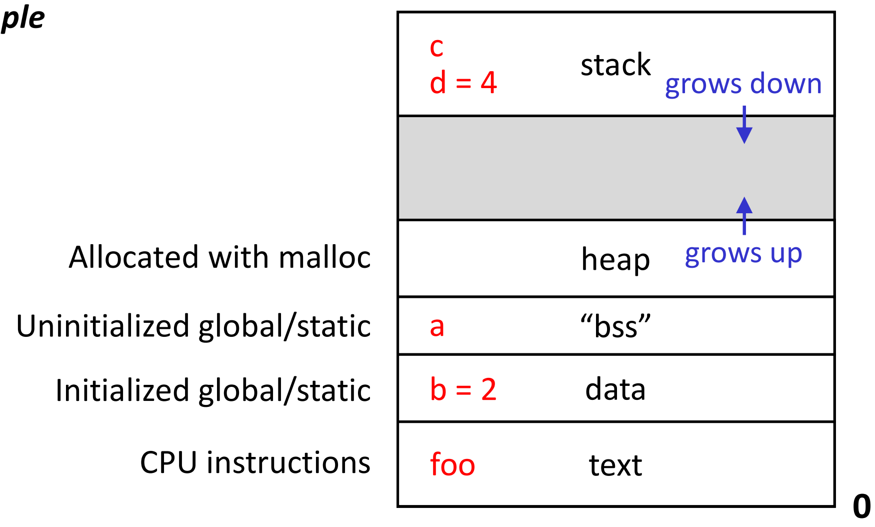
COMP 321

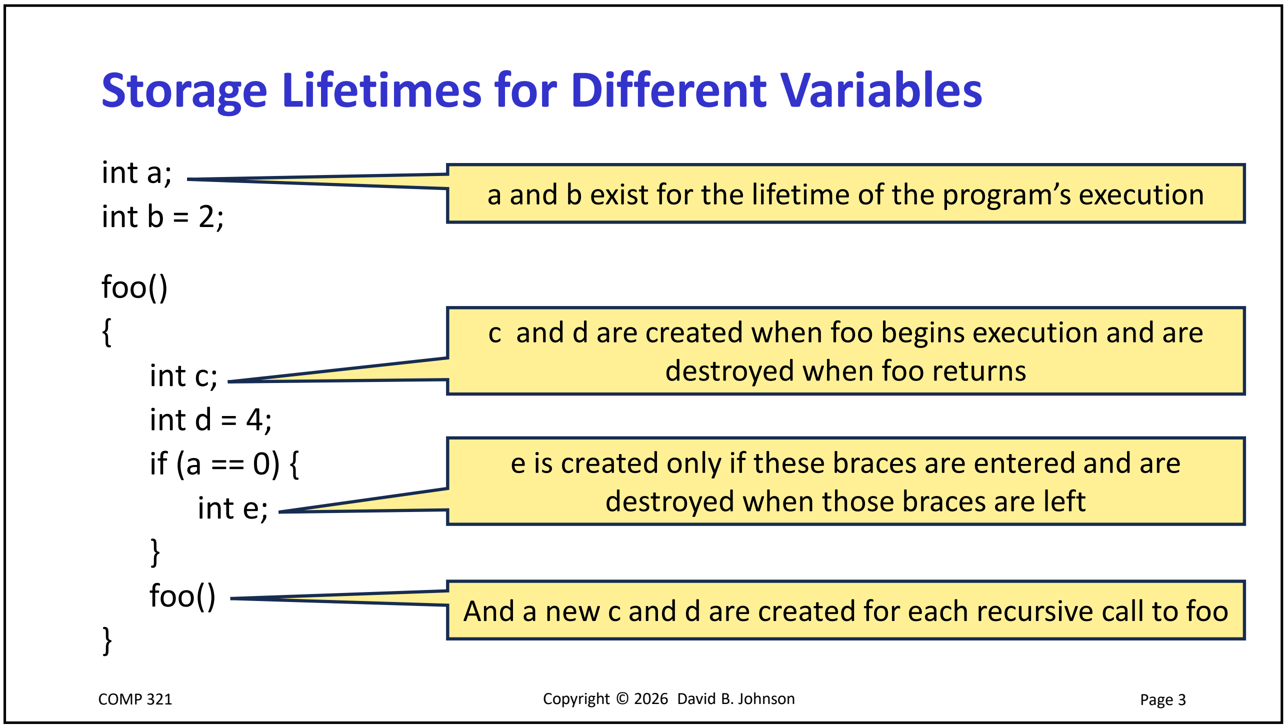
Dave Johnson

Review: A Process Address Space

A simplified example

```
int a;
int b = 2;

foo()
{
    int c;
    int d = 4;
    foo()
}
```





Storage Lifetimes for Different Variables

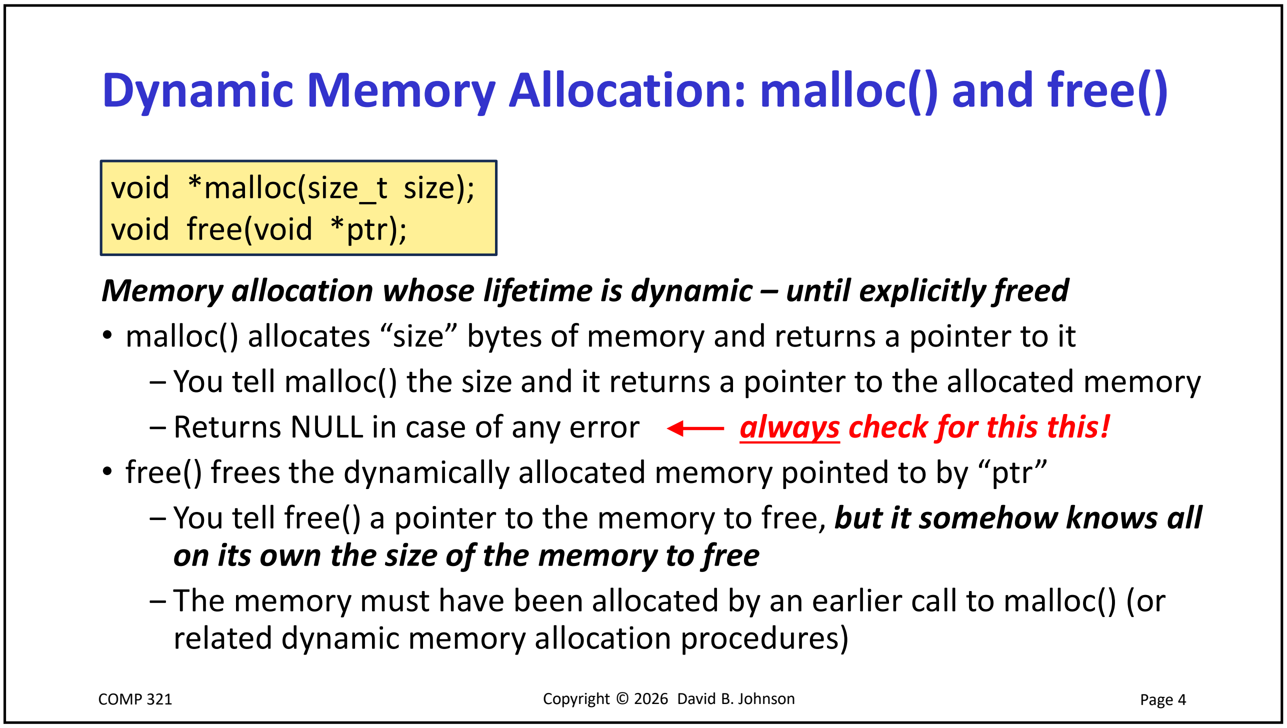
```
int a;
int b = 2;

foo()
{
    int c;
    int d = 4;
    if (a == 0) {
        int e;
    }
    foo()
}
```

a and b exist for the lifetime of the program’s execution

c and d are created when foo begins execution and are destroyed when foo returns

e is created only if these braces are entered and are destroyed when those braces are left

And a new c and d are created for each recursive call to foo

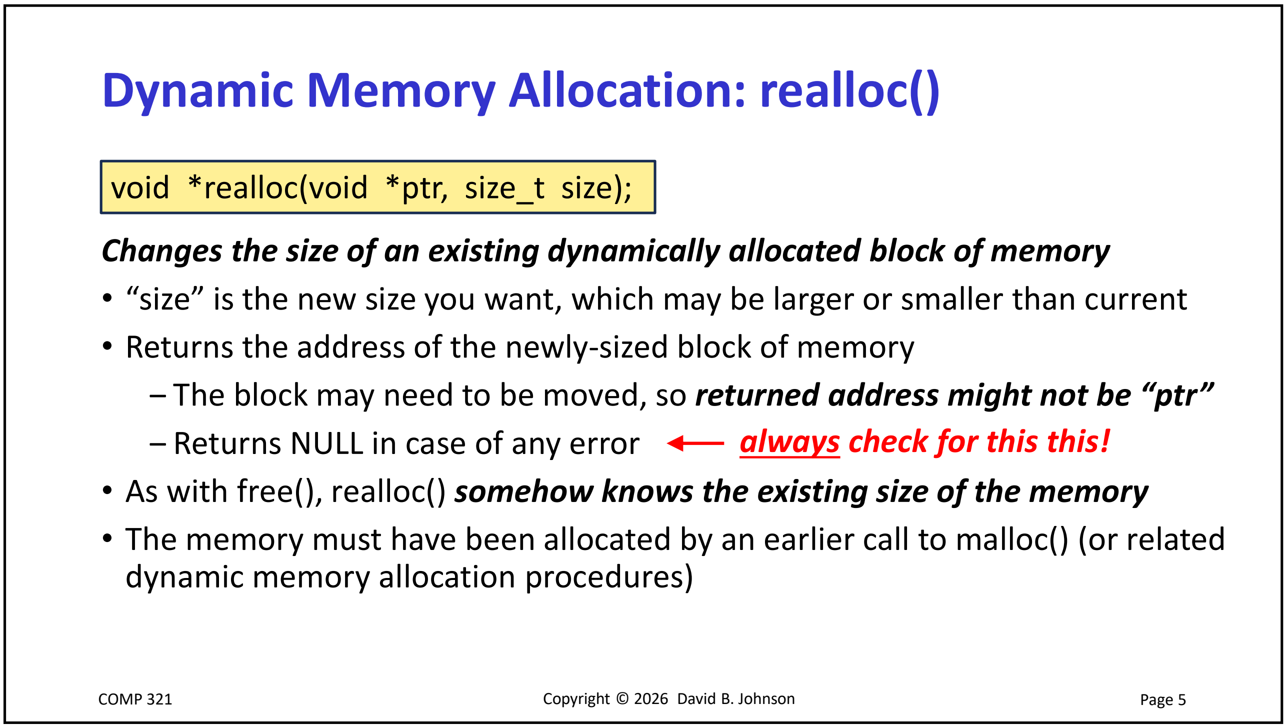


Dynamic Memory Allocation: malloc() and free()

```
void  *malloc(size_t  size);
void  free(void  *ptr);
```

Memory allocation whose lifetime is dynamic – until explicitly freed

- malloc() allocates “size” bytes of memory and returns a pointer to it
  - You tell malloc() the size and it returns a pointer to the allocated memory
  - Returns NULL in case of any error ← ***always check for this this!***
- free() frees the dynamically allocated memory pointed to by “ptr”
  - You tell free() a pointer to the memory to free, ***but it somehow knows all on its own the size of the memory to free***
  - The memory must have been allocated by an earlier call to malloc() (or related dynamic memory allocation procedures)

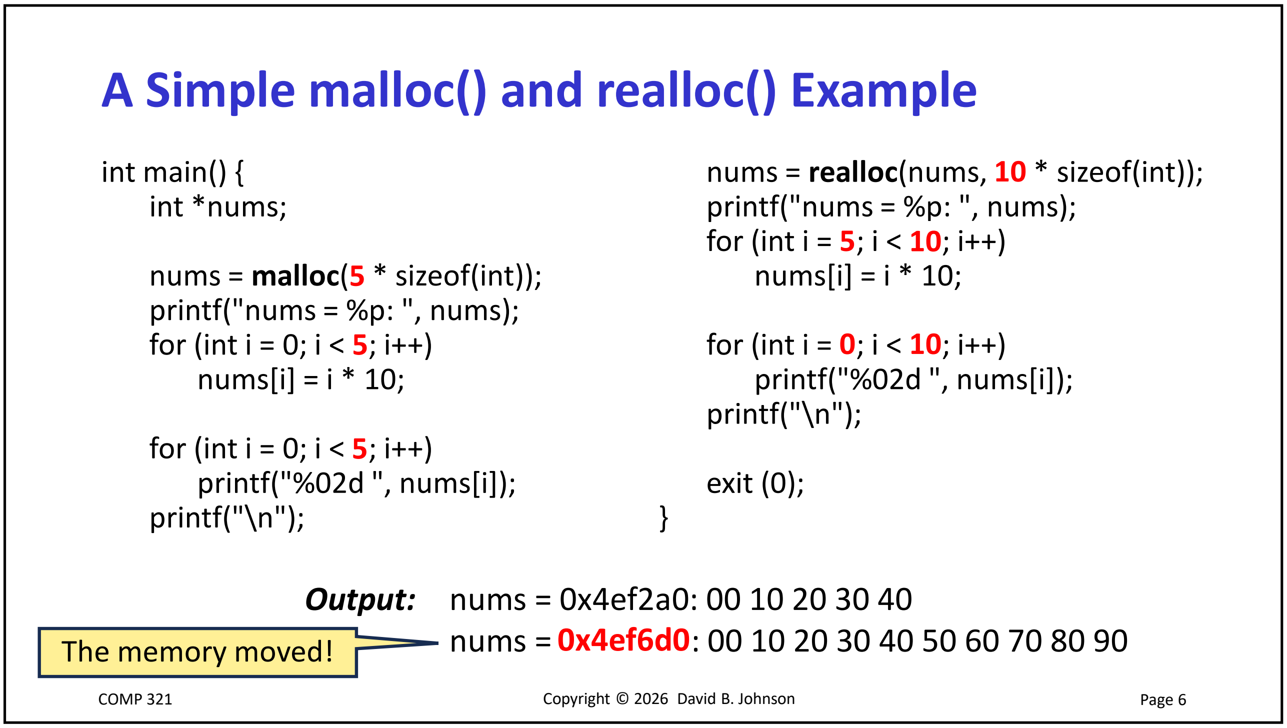


Dynamic Memory Allocation: realloc()

```
void  *realloc(void  *ptr,  size_t  size);
```

Changes the size of an existing dynamically allocated block of memory

- “size” is the new size you want, which may be larger or smaller than current
- Returns the address of the newly-sized block of memory
  - The block may need to be moved, so ***returned address might not be “ptr”***
  - Returns NULL in case of any error ← ***always check for this this!***
- As with free(), realloc() ***somehow knows the existing size of the memory***
- The memory must have been allocated by an earlier call to malloc() (or related dynamic memory allocation procedures)



A Simple malloc() and realloc() Example

```
int main() {
    int *nums;

    nums = malloc(5 * sizeof(int));
    printf("nums = %p: ", nums);
    for (int i = 0; i < 5; i++)
        nums[i] = i * 10;

    for (int i = 0; i < 5; i++)
        printf("%02d ", nums[i]);
    printf("\n");

    nums = realloc(nums, 10 * sizeof(int));
    printf("nums = %p: ", nums);
    for (int i = 5; i < 10; i++)
        nums[i] = i * 10;

    for (int i = 0; i < 10; i++)
        printf("%02d ", nums[i]);
    printf("\n");

    exit (0);
}
```

Output:

```
nums = 0x4ef2a0: 00 10 20 30 40
nums = 0x4ef6d0: 00 10 20 30 40 50 60 70 80 90
```

The memory moved!

Dynamic Memory Allocation: calloc()

```
void  *calloc(size_t  nmemb,  size_t  size);
```

An alternative interface for dynamic memory allocation

- Allocates “nmemb” (number of members), each of size “size” bytes
  - Basically, like allocating an array
  - where each element in the array is of size “size” bytes
  - and there are “nmemb” elements in the array
  - Example: calloc(100, sizeof(int))   allocates an array of 100 ints
- Returns the address of the newly-sized block of memory
  - Returns NULL in case of any error ← ***always check for this this!***

The Initial Values of the Allocated Memory

For malloc()

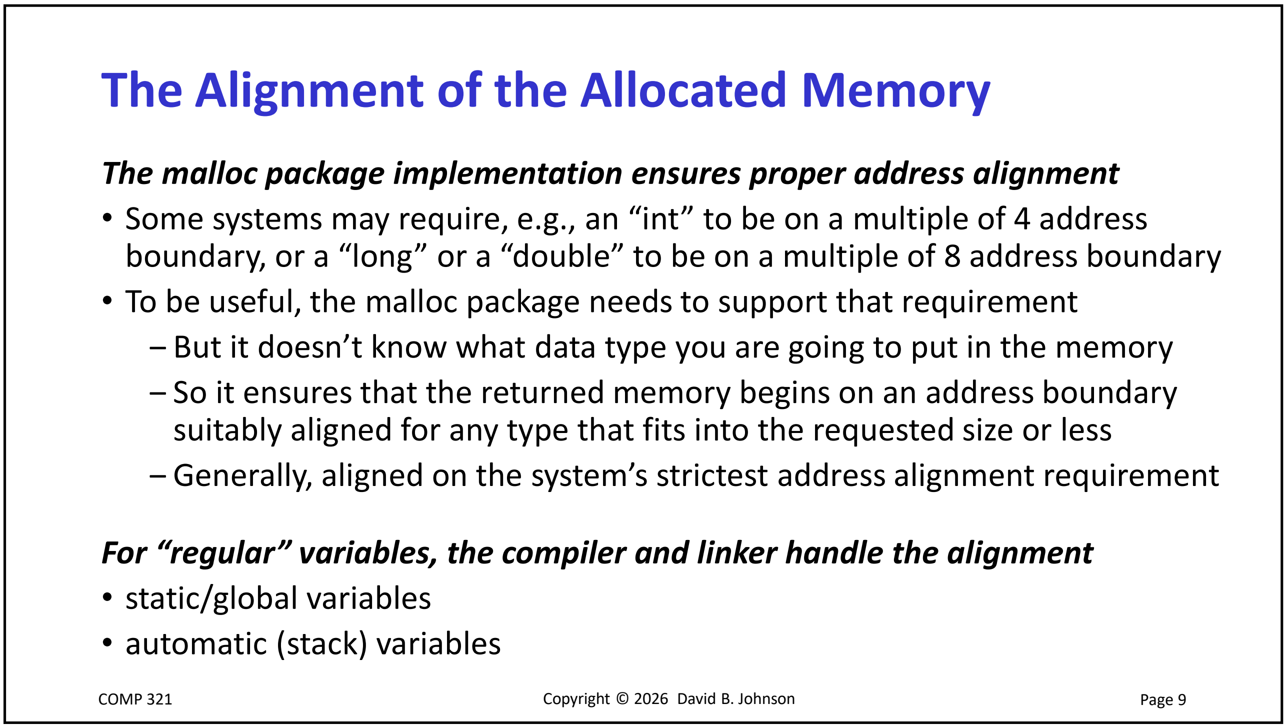
- The allocated memory is ***uninitialized***
- May contain any value – often not equal to 0

For realloc()

- The contents of the block is preserved, even if realloc() moves the block
- But if the block is larger, the additional new bytes at the end are ***uninitialized***

For calloc()

- The allocated memory is ***all initialized to 0***



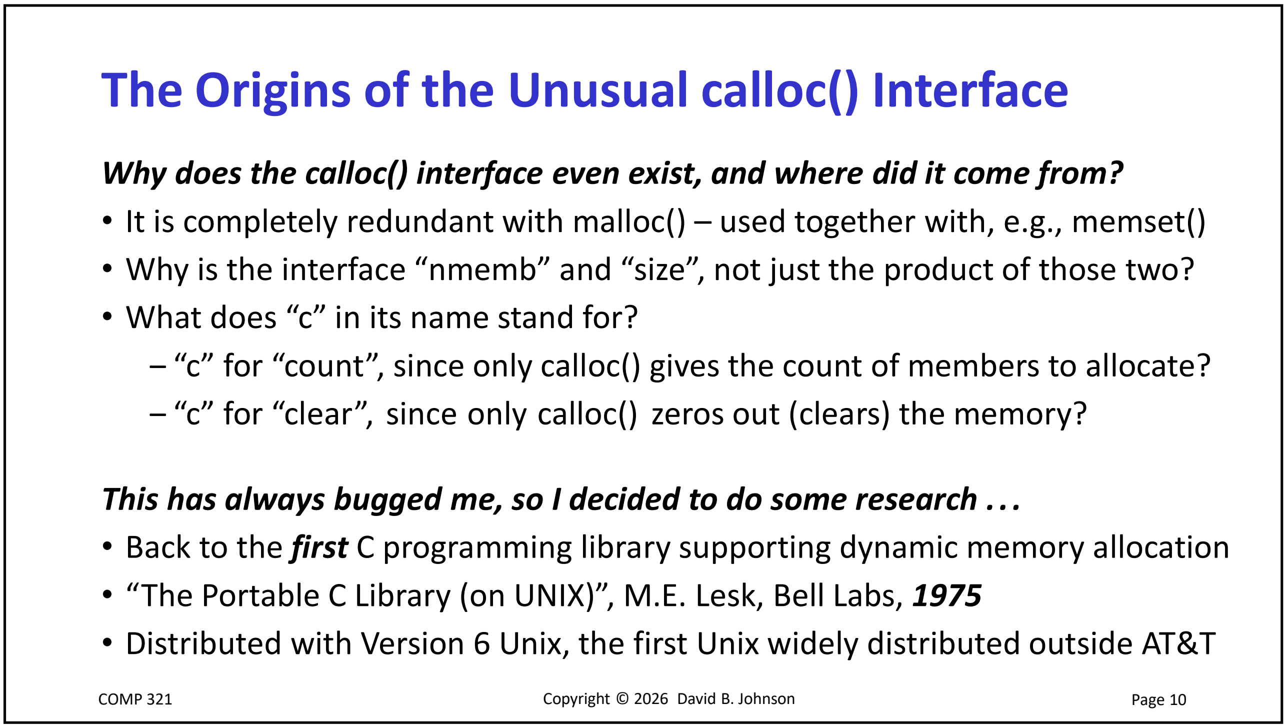
The Alignment of the Allocated Memory

The malloc package implementation ensures proper address alignment

- Some systems may require, e.g., an “int” to be on a multiple of 4 address boundary, or a “long” or a “double” to be on a multiple of 8 address boundary
- To be useful, the malloc package needs to support that requirement
  - But it doesn’t know what data type you are going to put in the memory
  - So it ensures that the returned memory begins on an address boundary suitably aligned for any type that fits into the requested size or less
  - Generally, aligned on the system’s strictest address alignment requirement

For “regular” variables, the compiler and linker handle the alignment

- static/global variables
- automatic (stack) variables



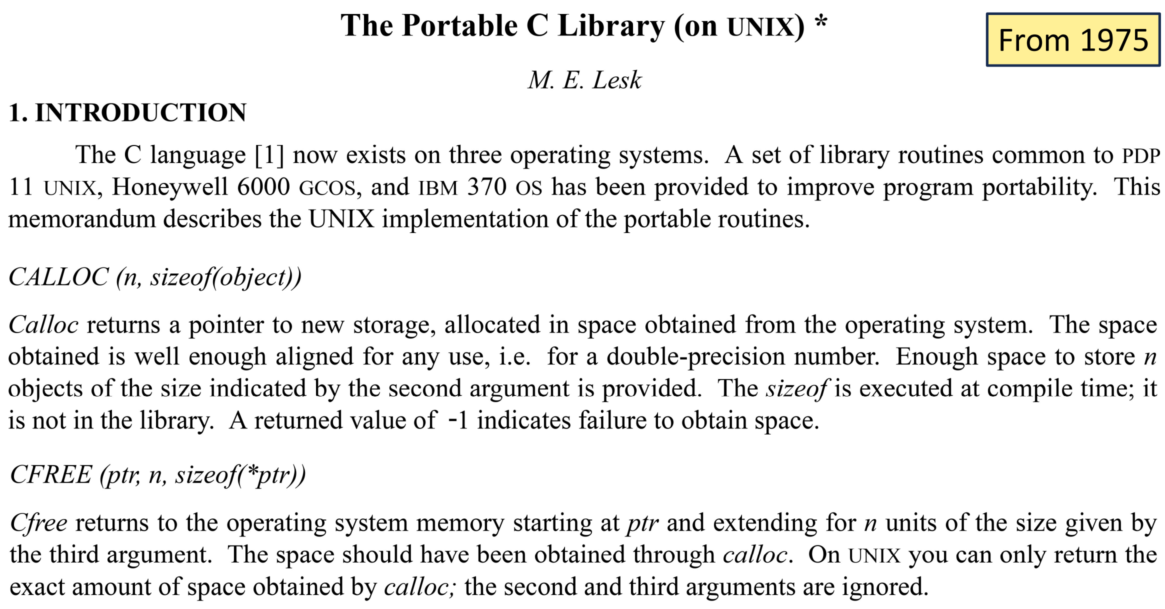
The Origins of the Unusual calloc() Interface

Why does the calloc() interface even exist, and where did it come from?

- It is completely redundant with malloc() – used together with, e.g., memset()
- Why is the interface “nmemb” and “size”, not just the product of those two?
- What does “c” in its name stand for?
  - “c” for “count”, since only calloc() gives the count of members to allocate?
  - “c” for “clear”, since only calloc() zeros out (clears) the memory?

This has always bugged me, so I decided to do some research . . .

- Back to the ***first*** C programming library supporting dynamic memory allocation
- “The Portable C Library (on UNIX)”, M.E. Lesk, Bell Labs, ***1975***
- Distributed with Version 6 Unix, the first Unix widely distributed outside AT&T



From 1975

The Portable C Library (on UNIX) *

M. E. Lesk

1. INTRODUCTION

The C language [1] now exists on three operating systems. A set of library routines common to PDP 11 UNIX, Honeywell 6000 GCOS, and IBM 370 OS has been provided to improve program portability. This memorandum describes the UNIX implementation of the portable routines.

CALLOC (n, sizeof(object))

Calloc returns a pointer to new storage, allocated in space obtained from the operating system. The space obtained is well enough aligned for any use, i.e. for a double-precision number. Enough space to store *n* objects of the size indicated by the second argument is provided. The *sizeof* is executed at compile time; it is not in the library. A returned value of -1 indicates failure to obtain space.

*CFREE (ptr, n, sizeof(*ptr))*

Cfree returns to the operating system memory starting at *ptr* and extending for *n* units of the size given by the third argument. The space should have been obtained through *calloc*. On UNIX you can only return the exact amount of space obtained by *calloc;* the second and third arguments are ignored.

Where Does the Memory Actually Come From?

The malloc package just hands out and keeps track of blocks of memory

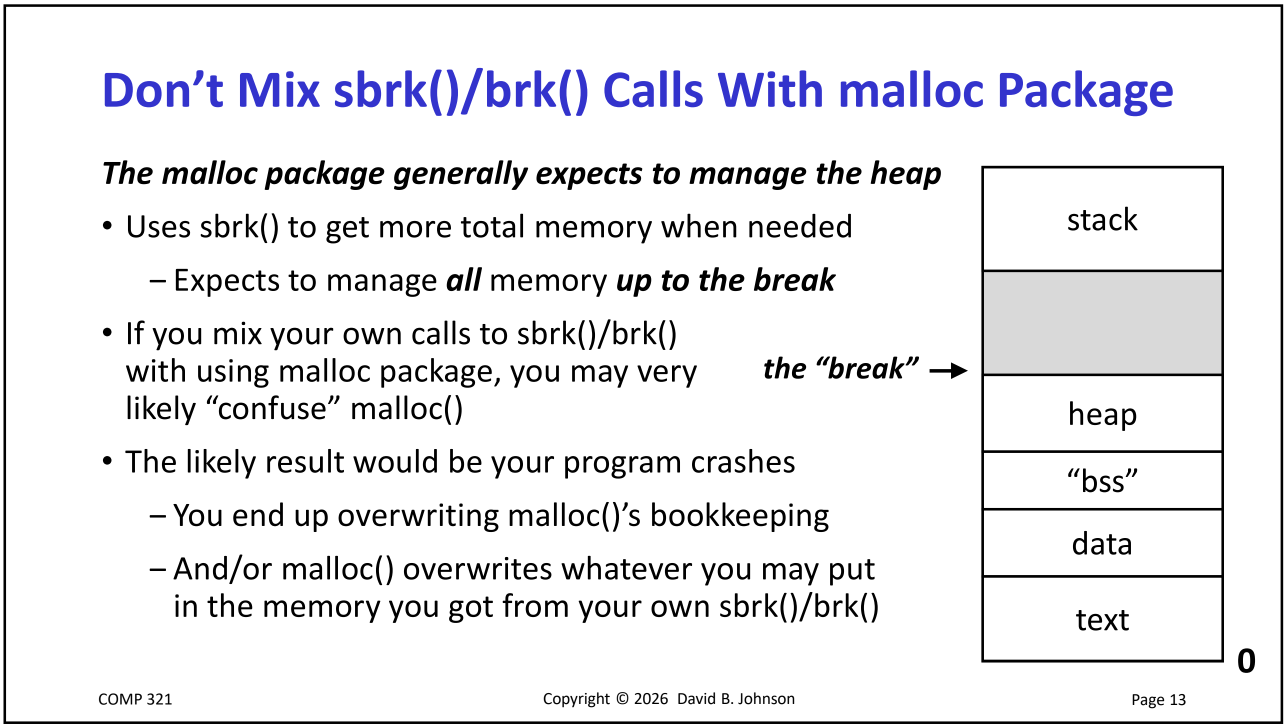
- These procedures don’t “create” the memory

The original answer, the brk() kernel call (still used in some implementations)

```
int  brk(void  *addr);
void  *sbrk(intptr_t  increment);
```

a library call that calls brk()

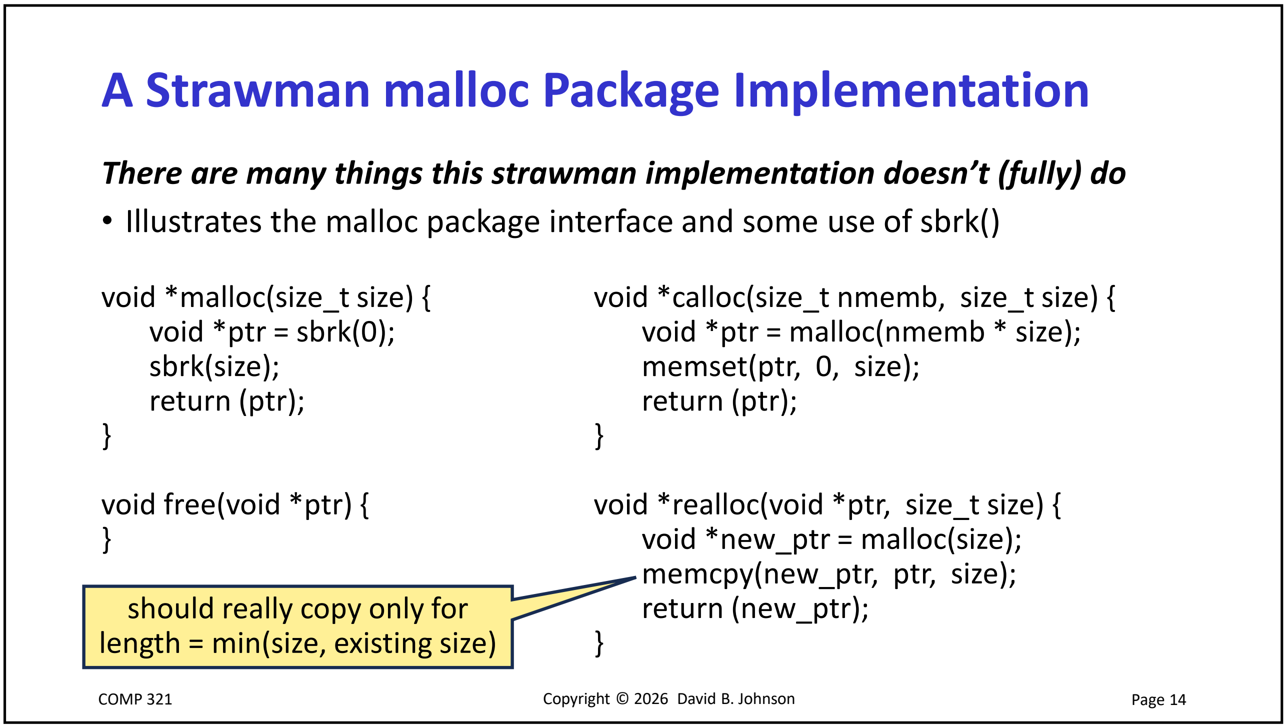
- A process’s ***break*** marks the end of its heap = address of first byte beyond heap
- brk() sets the break address to “addr”
- sbrk() can move the break up or down, and returns the new break address
  - Example: sbrk(0) returns the current break address
  - Example: malloc() can call sbrk(4096) to get 4 kB more memory from the operating system to then carve up for future malloc() requests as needed



Don’t Mix sbrk()/brk() Calls With malloc Package

The malloc package generally expects to manage the heap

- Uses sbrk() to get more total memory when needed
  - Expects to manage ***all*** memory ***up to the break***
- If you mix your own calls to sbrk()/brk() with using malloc package, you may very likely “confuse” malloc()
- The likely result would be your program crashes
  - You end up overwriting malloc()’s bookkeeping
  - And/or malloc() overwrites whatever you may put in the memory you got from your own sbrk()/brk()



A Strawman malloc Package Implementation

There are many things this strawman implementation doesn’t (fully) do

- Illustrates the malloc package interface and some use of sbrk()

```
void *malloc(size_t size) {
     void *ptr = sbrk(0);
     sbrk(size);
     return (ptr);
}

void free(void *ptr) {
}
```

```
void *calloc(size_t nmemb,  size_t size) {
     void *ptr = malloc(nmemb * size);
     memset(ptr,  0,  size);
     return (ptr);
}

void *realloc(void *ptr,  size_t size) {
     void *new_ptr = malloc(size);
     memcpy(new_ptr,  ptr,  size);
     return (new_ptr);
}
```

should really copy only for length = min(size, existing size)

Some Needed malloc Package Bookkeeping

Keeps track of allocated blocks

- At least so that it can know the size of the block on a future free() of it

Keeps track of free blocks (i.e., some kind of a free list)

- Memory that it is managing that isn’t in use for any current allocation
  - Including memory returned from earlier free() calls so it can be reused
- When some new allocation request is made
  - Which existing free block ***should*** be used (or which is ***best*** to use)?
  - And how does the malloc package find that existing free block?
- When some existing allocated block is freed
  - What to do with that block vs. other existing free blocks being managed?

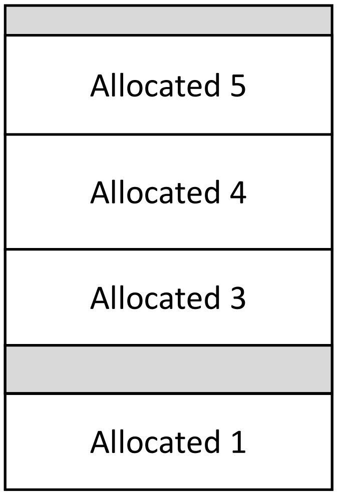
A Problem of Two Different Kinds

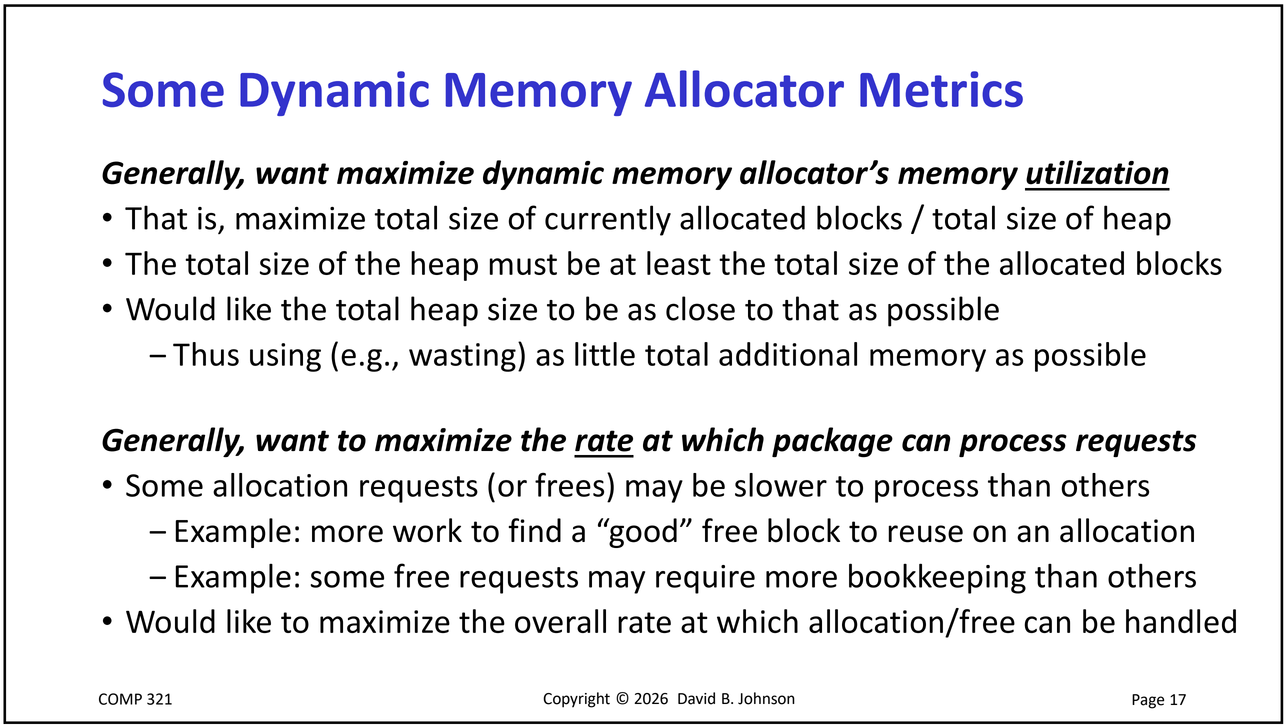
Internal fragmentation

- Unused space ***inside*** an existing allocated block
  - Example: Allocated size was rounded up to some required allocation granularity
  - Example: Initially allocated more space than needed in case the use needs to grow

External fragmentation

- Unusable space ***between*** allocated blocks
  - Enough total space but no single contiguous space is large enough for some new request





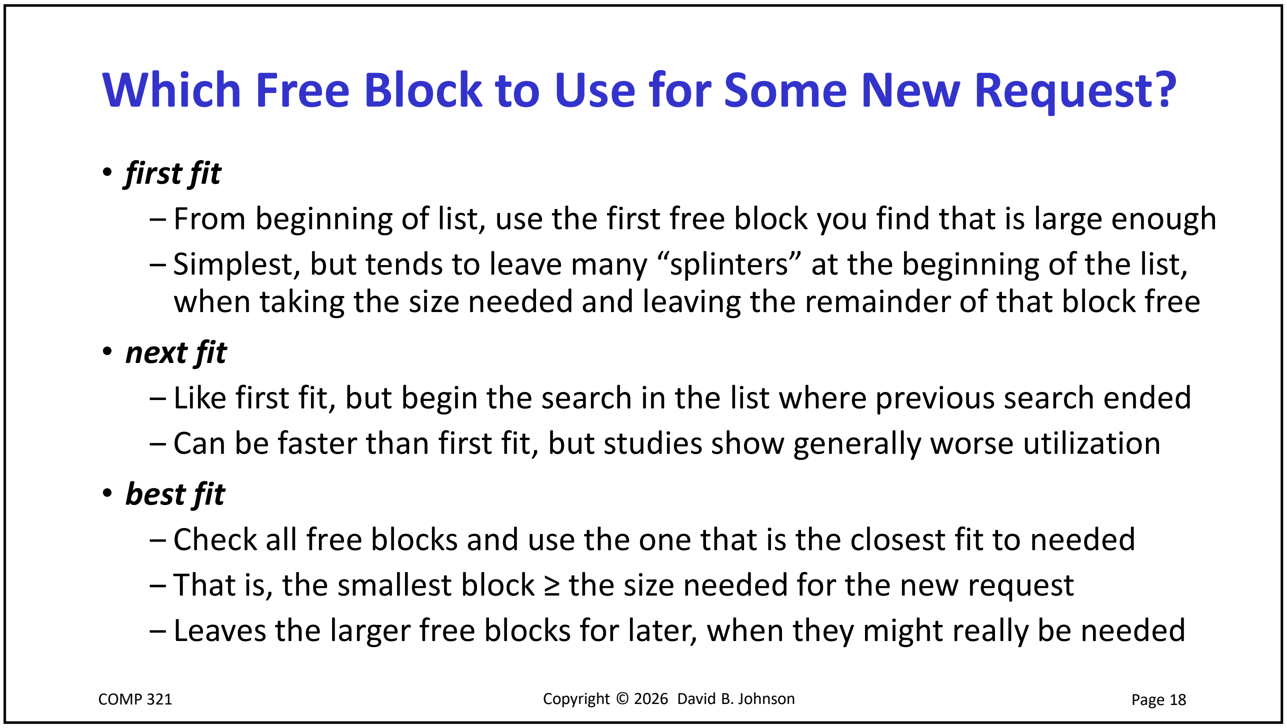
Some Dynamic Memory Allocator Metrics

Generally, want maximize dynamic memory allocator's memory utilization

- That is, maximize total size of currently allocated blocks / total size of heap
- The total size of the heap must be at least the total size of the allocated blocks
- Would like the total heap size to be as close to that as possible
  - Thus using (e.g., wasting) as little total additional memory as possible

Generally, want to maximize the rate at which package can process requests

- Some allocation requests (or frees) may be slower to process than others
  - Example: more work to find a "good" free block to reuse on an allocation
  - Example: some free requests may require more bookkeeping than others
- Would like to maximize the overall rate at which allocation/free can be handled



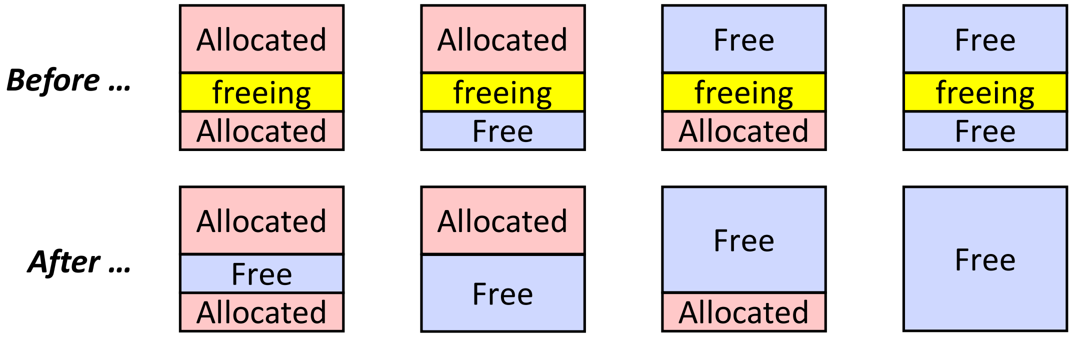
Which Free Block to Use for Some New Request?

- ***first fit***
  - From beginning of list, use the first free block you find that is large enough
  - Simplest, but tends to leave many "splinters" at the beginning of the list, when taking the size needed and leaving the remainder of that block free
- ***next fit***
  - Like first fit, but begin the search in the list where previous search ended
  - Can be faster than first fit, but studies show generally worse utilization
- ***best fit***
  - Check all free blocks and use the one that is the closest fit to needed
  - That is, the smallest block ≥ the size needed for the new request
  - Leaves the larger free blocks for later, when they might really be needed

Adding a Block Being Freed Back Into the Free List

To keep down external fragmentation, “coalesce” it with neighboring free blocks

- Consider the four possible cases for the block in memory before and after



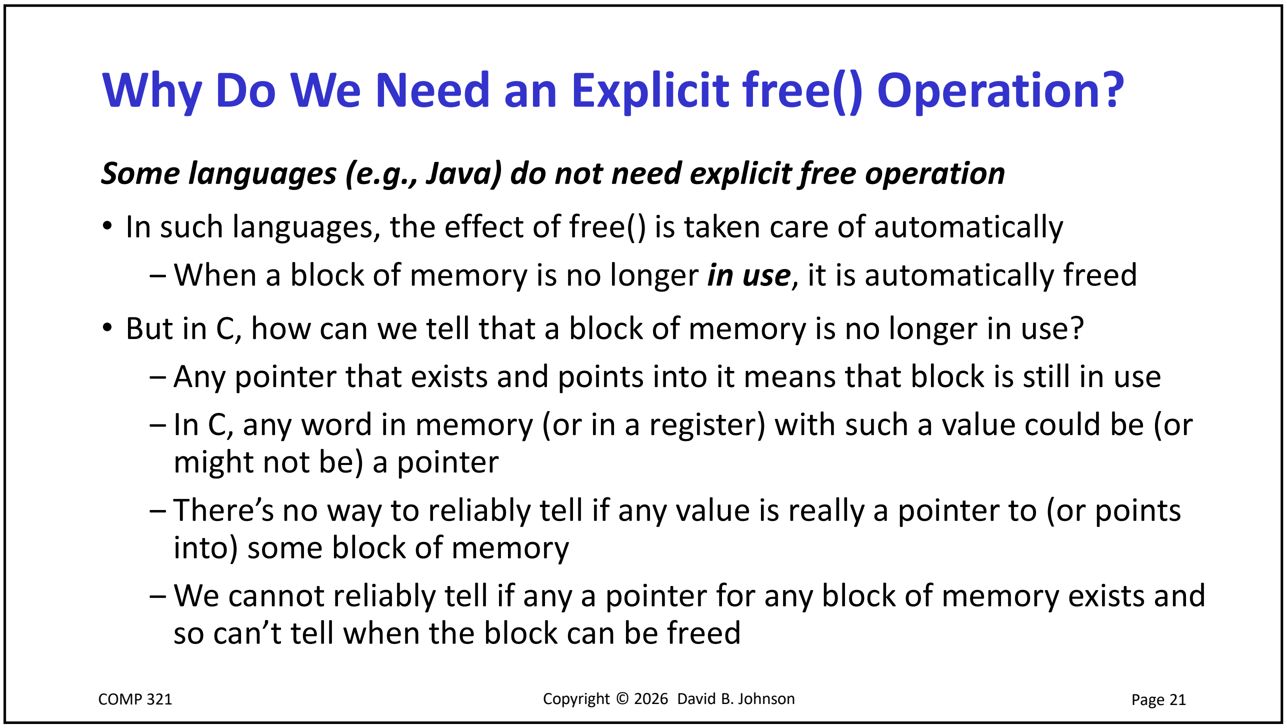
Before …	Allocated freeing Allocated	Allocated freeing Free	Free freeing Allocated	Free freeing Free
After …	Allocated Free Allocated	Allocated Free	Free Allocated	Free

- How easy or hard to do in practice depends on how the bookkeeping is done

Does free() Ever Give Memory Back to the Kernel?

As memory is freed, it is difficult for malloc package give memory back to OS

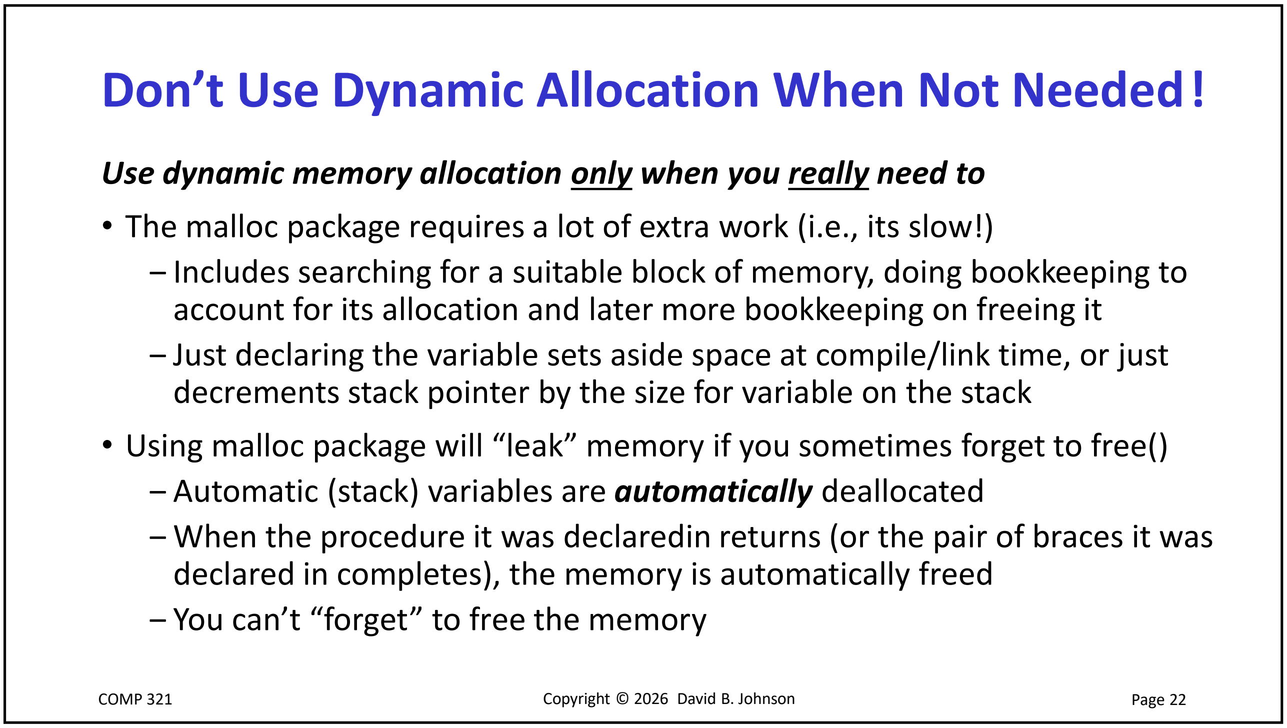
- As more total memory is needed, malloc package can get it from the kernel
- But it is difficult/impossible to give memory back to the kernel
  - malloc package can call brk(addr) to trim the heap size down do addr
  - But this can only be done if ***all*** memory at addresses ≥ addr is ***all*** free
  - ***and*** is not needed as part of the malloc package’s internal bookkeeping
- This could occur but may not be very likely
  - Far easier to not give the memory back to the kernel
  - and continue instead to manage it for future allocation requests
- And all memory is automatically freed when the process terminates



Why Do We Need an Explicit free() Operation?

Some languages (e.g., Java) do not need explicit free operation

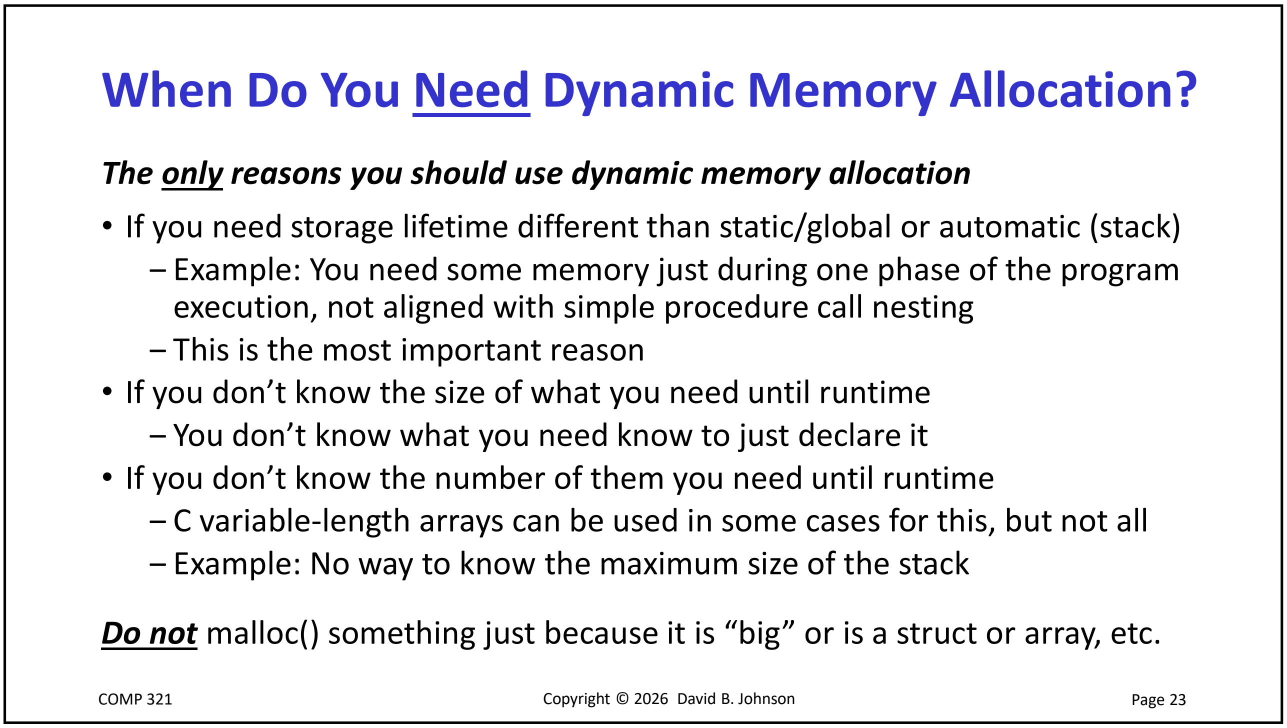
- In such languages, the effect of free() is taken care of automatically
  - When a block of memory is no longer ***in use***, it is automatically freed
- But in C, how can we tell that a block of memory is no longer in use?
  - Any pointer that exists and points into it means that block is still in use
  - In C, any word in memory (or in a register) with such a value could be (or might not be) a pointer
  - There's no way to reliably tell if any value is really a pointer to (or points into) some block of memory
  - We cannot reliably tell if any a pointer for any block of memory exists and so can't tell when the block can be freed



Don't Use Dynamic Allocation When Not Needed!

Use dynamic memory allocation <u>only</u> when you <u>really</u> need to

- The malloc package requires a lot of extra work (i.e., its slow!)
  - Includes searching for a suitable block of memory, doing bookkeeping to account for its allocation and later more bookkeeping on freeing it
  - Just declaring the variable sets aside space at compile/link time, or just decrements stack pointer by the size for variable on the stack
- Using malloc package will "leak" memory if you sometimes forget to free()
  - Automatic (stack) variables are ***automatically*** deallocated
  - When the procedure it was declaredin returns (or the pair of braces it was declared in completes), the memory is automatically freed
  - You can't "forget" to free the memory



When Do You <u>Need</u> Dynamic Memory Allocation?

The <u>only</u> reasons you should use dynamic memory allocation

- If you need storage lifetime different than static/global or automatic (stack)
  - Example: You need some memory just during one phase of the program execution, not aligned with simple procedure call nesting
  - This is the most important reason
- If you don't know the size of what you need until runtime
  - You don't know what you need know to just declare it
- If you don't know the number of them you need until runtime
  - C variable-length arrays can be used in some cases for this, but not all
  - Example: No way to know the maximum size of the stack

<u>Do not</u> malloc() something just because it is "big" or is a struct or array, etc.