University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Dynamic Resizing

Programmers Rarely Know How to Size an Array

Allocation of arrays
- at compile time (**static allocation**)
- forces programmer to choose array size.

Often, there's no good way to choose.

For example,
- how many players do we need
- for ***First International Blocky Server***?

10? 10,000? 10,000,000?

Dynamic Resizing Grows Array to Fit Demand

One solution to this dilemma
is called **dynamic resizing**:
- **Start with 10** players.
- If we **need > 10, change to 20**.
- If we **need > 20, change to 40**.
- And so forth.

Each time we grow the array
- existing **players must be copied**
- to the new array.

How Much Copying is Needed for Dynamic Resizing?

Before we see how it works,
- it's worth asking:
- **how expensive is the copying?**

We can bound it:
- if we have **N players in the array**,
- the **last copy copied at most N** players,*
- and the **previous copy** copied **at most N/2**,
- and **the one before that, at most N/4**.

*Technically (N – 1), but we're finding an upper bound anyway.

Resizing Copies at Most Twice the Number of Players

In other words,

- with **N** players, we copy at most
- **$N (1 + \frac{1}{2} + \frac{1}{4} + \ldots)$** players.

In the infinite limit,*

- we have **at most $2N$** players copied, or
- **2 copies per player in the array**.

That's not too bad.

*The number of times that the array grows is finite, but, again, we want a bound.

How Much Space is Wasted with Dynamic Resizing?

What about wasted space?

Dynamic resizing always

- multiplies the size of the array,
- so there's usually a lot of empty space.

Note that if one

- extends the array by adding a fixed amount,
- copying cost is quadratic in **N**, not linear.

Assume Uniform Distribution Between Powers of 2

To answer, we **must make an assumption about** the **likelihood** of various values of **N**.

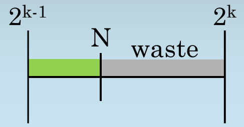
Let's do this:

- For simplicity, assume that we start with 1.
- Any value of **N** falls between two powers of 2: **$2^{k-1} < N \leq 2^k$** for some integer **k**.
- We **assume that values of N are evenly distributed in each such interval**.

Waste Space Needs to be Averaged

The figure to the right illustrates the problem.

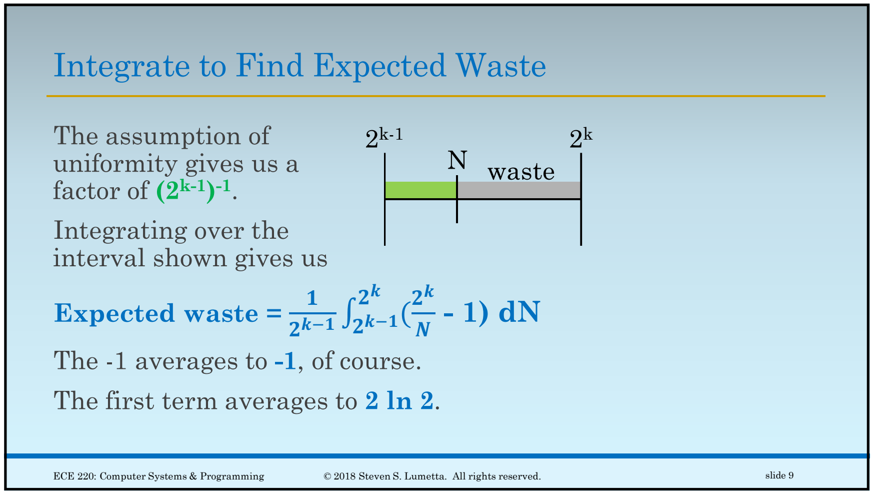
When **$2^{k-1} < N \leq 2^k$**, **we allocate 2^k to hold N**.



The amount of waste space is **$2^k - N$**.

In terms of the amount of space needed (**N**)

- dynamic resizing wastes another **$2^k/N - 1$**.
- Now we need to average.



Integrate to Find Expected Waste

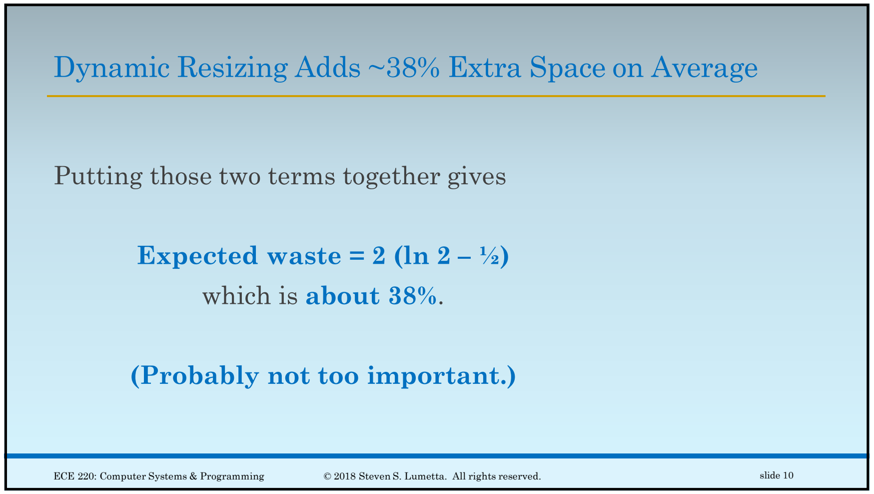
The assumption of uniformity gives us a factor of $(2^{k-1})^{-1}$.

Integrating over the interval shown gives us

$$\textbf{Expected waste} = \frac{1}{2^{k-1}}\int_{2^{k-1}}^{2^k}\left(\frac{2^k}{N} - 1\right) dN$$

The -1 averages to **-1**, of course.

The first term averages to **2 ln 2**.



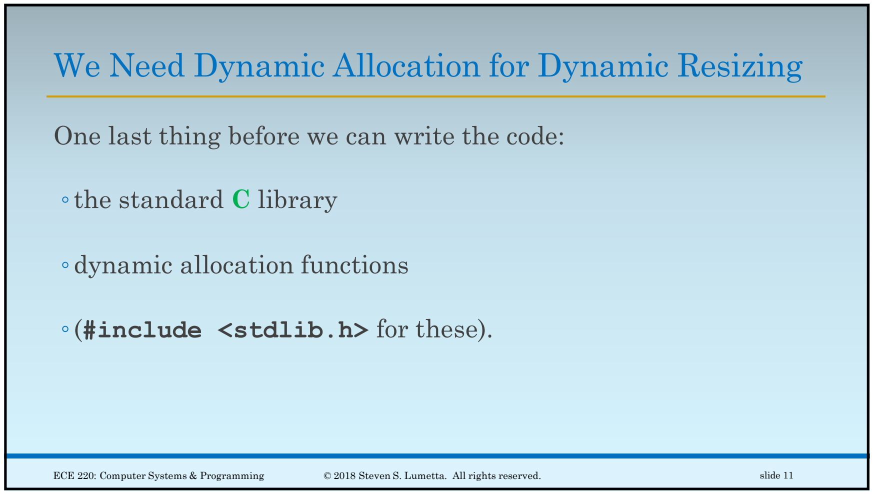
Dynamic Resizing Adds ~38% Extra Space on Average

Putting those two terms together gives

Expected waste = 2 (ln 2 – ½)

which is **about 38%**.

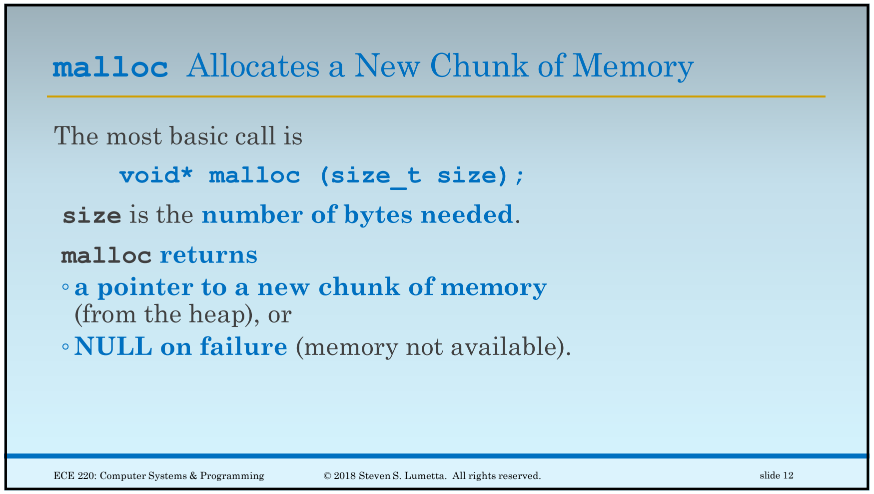
(Probably not too important.)



We Need Dynamic Allocation for Dynamic Resizing

One last thing before we can write the code:

- the standard **C** library
- dynamic allocation functions
- (`#include <stdlib.h>` for these).



`malloc` Allocates a New Chunk of Memory

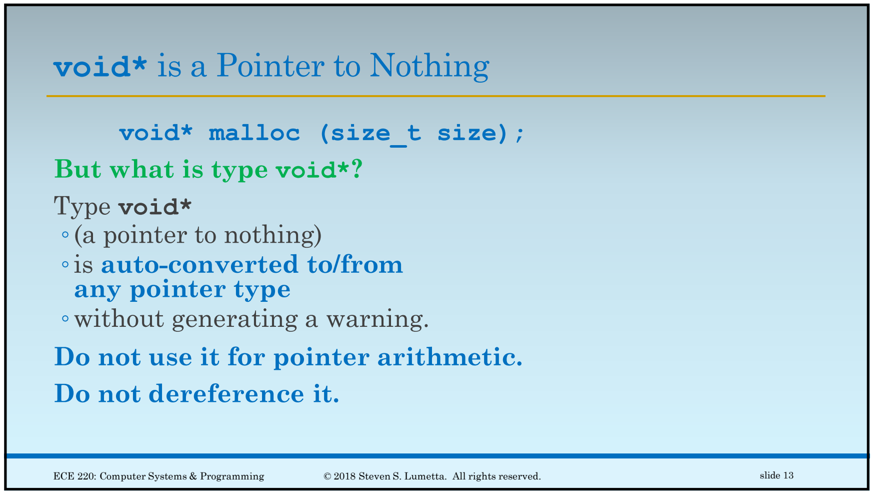
The most basic call is

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

`size` is the **number of bytes needed**.

`malloc` **returns**

- **a pointer to a new chunk of memory** (from the heap), or
- **NULL on failure** (memory not available).



`void*` is a Pointer to Nothing

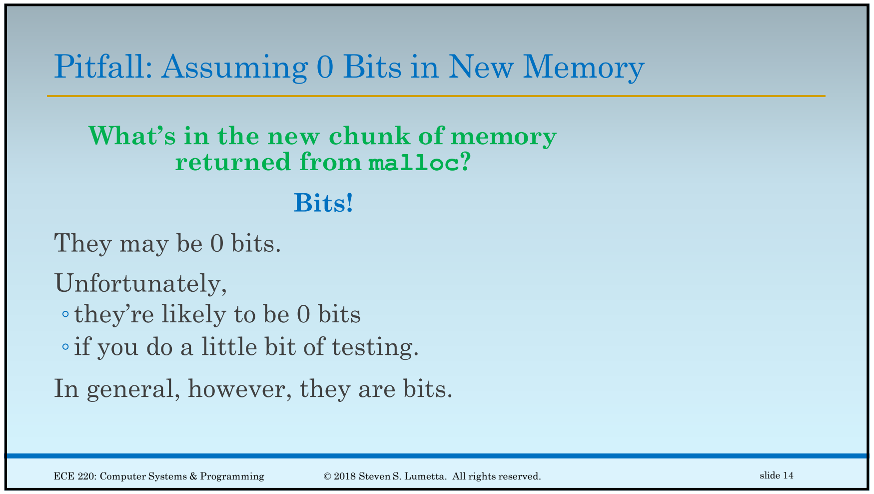
```
void* malloc (size_t size);
```

But what is type `void*`?

Type **`void*`**
- (a pointer to nothing)
- is **auto-converted to/from any pointer type**
- without generating a warning.

Do not use it for pointer arithmetic.

Do not dereference it.



Pitfall: Assuming 0 Bits in New Memory

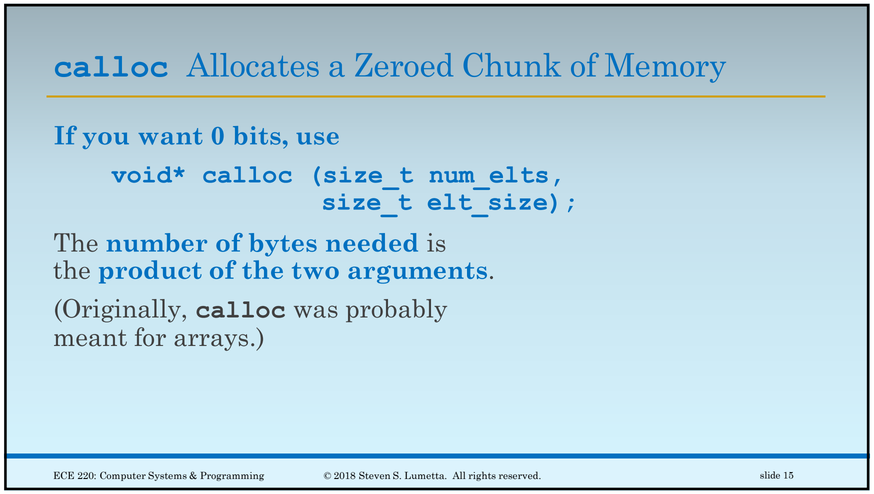
What's in the new chunk of memory returned from `malloc`?

Bits!

They may be 0 bits.

Unfortunately,
- they're likely to be 0 bits
- if you do a little bit of testing.

In general, however, they are bits.



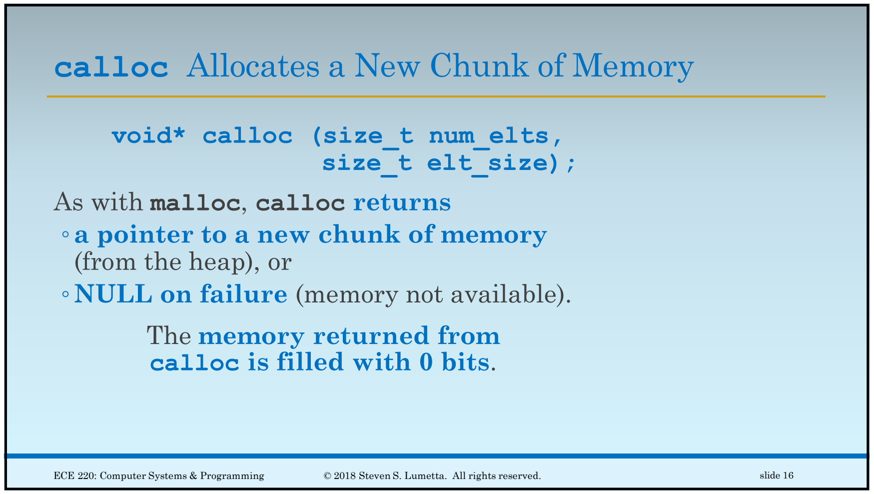
`calloc` Allocates a Zeroed Chunk of Memory

If you want 0 bits, use

```
void* calloc (size_t num_elts,
              size_t elt_size);
```

The **number of bytes needed** is the **product of the two arguments**.

(Originally, **`calloc`** was probably meant for arrays.)



`calloc` Allocates a New Chunk of Memory

```
void* calloc (size_t num_elts,
              size_t elt_size);
```

As with **`malloc`**, **`calloc` returns**
- **a pointer to a new chunk of memory** (from the heap), or
- **NULL on failure** (memory not available).

The **memory returned from `calloc` is filled with 0 bits**.

`realloc` Resizes a Chunk of Memory

The third function is

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

This function is used to **change the size of an allocated block of memory**.

`size` is the **new number of bytes needed**.

`ptr` must be a dynamically allocated block (a value returned from `malloc`, `calloc`, or `realloc`).

`realloc` Returns a Chunk of Memory

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

`realloc` attempts
- to grow/shrink the block,
- as requested.

Caller need not know (nor pass) the original size.

As with `malloc` and `calloc`, `realloc` **returns**
- **a pointer to a chunk of memory** (from the heap), or
- **NULL on failure** (memory not available).

`realloc` Copies and Frees When Necessary

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

The **value returned** from `realloc` **may or may not be the same as `ptr`**.

If they differ,
- **data will be copied** from the old block to the new block,
- **and** the **old block will be freed**.

`free` Frees a Chunk of Memory

When your program is done with a block of dynamically allocated memory, you should call

```
void free (void* ptr);
```

`ptr` must be a dynamically allocated block (a value returned from `malloc`, `calloc`, or `realloc`).

Rules for Dynamic Allocation

Be sure to follow the rules when using dynamically allocated memory:

1. Do not read/write memory locations before/after a block.
2. Call **`free`** exactly once on each block.
3. Do not call **`free`** on any other pointer, including pointers into a block.
4. Do not access (read, nor write) a block after freeing it.

Overloading Meaning: `realloc`

I mentioned earlier that one should avoid overloading function meaning for no reason.

`realloc` is a good example.

Can't remember **`malloc`**'s name?

Just use **`realloc (NULL, size)`**!

Can't remember **`free`**'s name?

Just use **`realloc (ptr, 0)`**!

File Scope Variables for Dynamic Resizing

Now we're ready to write code.

We will need some file-scope variables:

```
static player_t* player_list = NULL;
static int32_t num_players = 0;
static int32_t max_players = 10;
```

`player_list` is the array. We cannot statically initialize it to a dynamic block.

Write `player_create` Using Dynamic Resizing

We will write

```
int32_t player_create (char* n,
    char* pswd, int32_t p_age,
    player_t** new_p);
```

which **uses dynamic resizing**

- to **find a free array element**,
- **initialize it** using **`player_init`**, and
- **return a pointer in `*new_p`**.

The return value is **1 for success, 0 for failure**.

Check Arguments Before Trying to Create Player

```
int32_t player_create
     (char* n, char* pswd,
      int32_t p_age,
      player_t** new_p)
{
     player_t* new_copy;

     ASSERT (NULL != name);
     ASSERT (NULL != pswd);
     ASSERT (NULL != new_p);
```

for **`player_init`**

to store new **`player_t*`**

Assert requirements.

First Case: First Time **`player_create`** is Called

```
if (NULL == player_list) {
     player_list =
          malloc (max_players *
                 sizeof (*player_list));
     if (NULL == player_list) {
          return 0;
     }
} else ...
```

No array yet?

Try to create array.

No memory? Give up.

Second Case: Array is Currently Full

```
if (max_players == num_players) {
     player_list = realloc
         (player_list,
          2 * max_players *
          sizeof (*player_list));
     // What's wrong with this code?

     max_player *= 2;
}
```

Array full?

Grow **`player_list`**.

Pitfall: Using **`realloc`** without a Temporary

If your code calls realloc this way:

```
ptr = realloc (ptr, new_size);
```

and **`realloc`** fails,

the address of your old block is gone!

Use a Temporary Variable When Calling realloc

Instead, create a temporary:

```
thing_t* new_copy;
new_copy = realloc (ptr, new_size);
if (NULL != new_copy) {
    ptr = new_copy;
}
```

Second Case: Array is Currently Full

```
if (max_players == num_players) {
    new_copy = realloc (player_list,
        2 * max_players *
        sizeof (*player_list));
    if (NULL == new_copy) {
        return 0;
    }
    max_player *= 2;
    player_list = new_copy;
}
```

Array full?

Second Case: Array is Currently Full

```
if (max_players == num_players) {
    new_copy = realloc (player_list,
        2 * max_players *
        sizeof (*player_list));
    if (NULL == new_copy) {
        return 0;
    }
    max_player *= 2;
    player_list = new_copy;
}
```

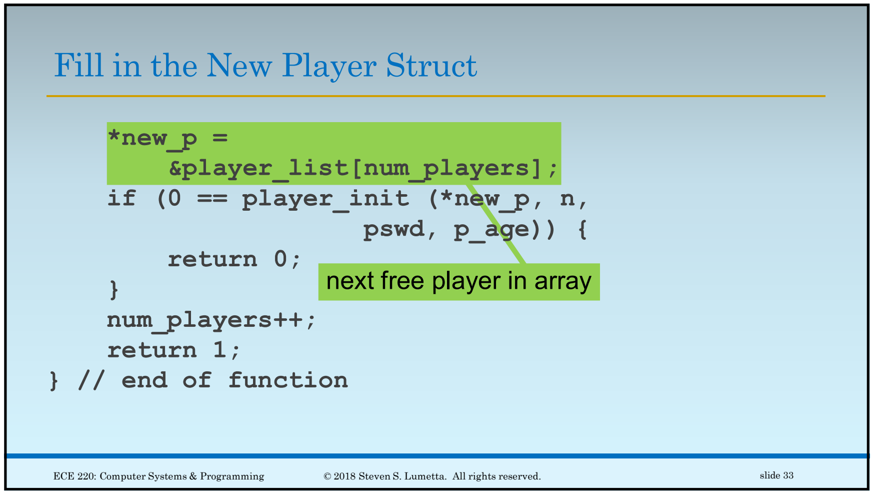
Grow **`player_list`**.

Second Case: Array is Currently Full

```
if (max_players == num_players) {
    new_copy = realloc (player_list,
        2 * max_players *
        sizeof (*player_list));
    if (NULL == new_copy) {
        return 0;
    }
    max_player *= 2;
    player_list = new_copy;
}
```

Out of memory? Give up.

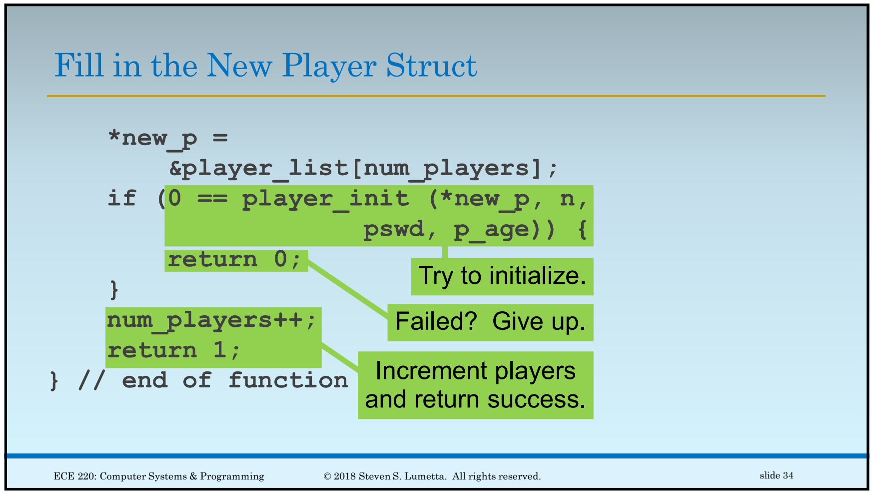
Update variables to reflect new size and place.



Fill in the New Player Struct

```
    *new_p =
        &player_list[num_players];
    if (0 == player_init (*new_p, n,
                         pswd, p_age)) {
        return 0;
    }
    num_players++;
    return 1;
} // end of function
```

next free player in array



Fill in the New Player Struct

```
    *new_p =
        &player_list[num_players];
    if (0 == player_init (*new_p, n,
                         pswd, p_age)) {
        return 0;
    }
    num_players++;
    return 1;
} // end of function
```

Try to initialize.

Failed? Give up.

Increment players and return success.