APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

Lecture 25
March 19, 2012

Today

- `realloc()`

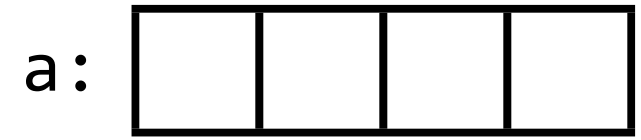
- Commenting
- Testing
- Debugging

Fixed Sizes

- Automatically allocated variables can't change size
- Until C99, array sizes needed to be fixed at compile time

Pre-C99

```
int a[4];
```



C99

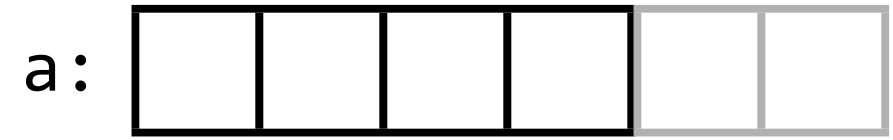
```
int n;
scanf("%d", &n);
int a[n];
```

Fixed Sizes

- Automatically allocated variables can't change size
- Until C99, array sizes needed to be fixed at compile time

Pre-C99

```
int a[4];
```



C99

```
int n;
scanf("%d", &n);
int a[n];
```

realloc()

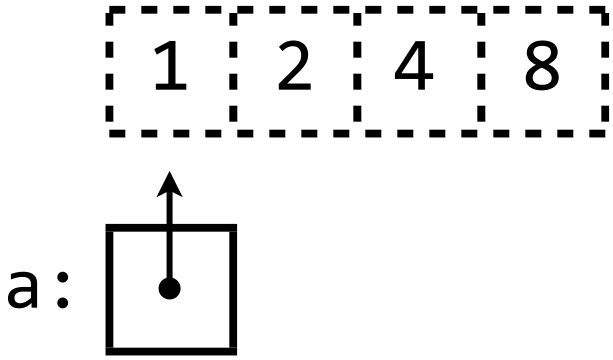
```
void *realloc(void *p, size_t size);
```

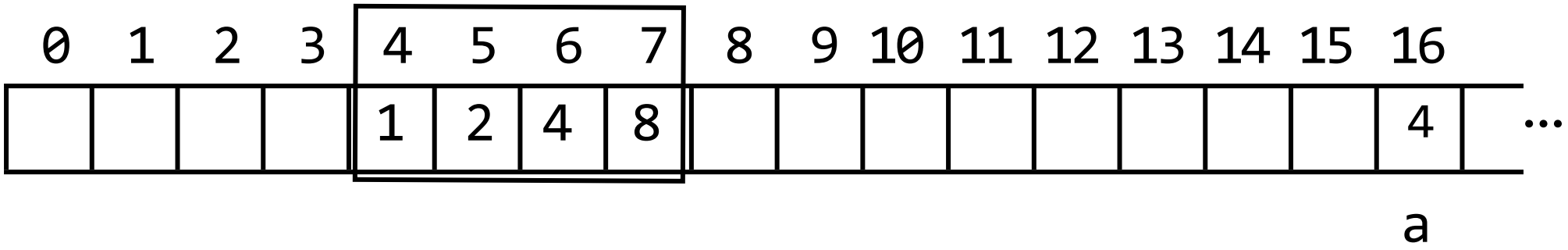
for now think of it as `int`

- Resizes a `malloc()`ed chunk of memory
- Can increase or decrease the size
- Returns a pointer to the resized memory
- `p = NULL` is equivalent to `malloc()`
- `size = 0` is equivalent to `free()`

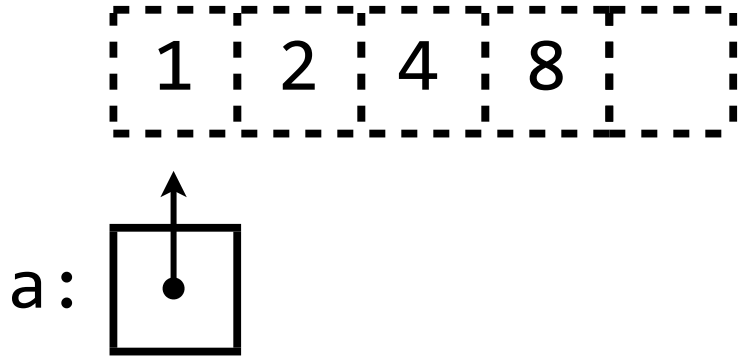
```
int *a = malloc(4 * sizeof(int));
a = realloc(a, 5 * sizeof(int));
```

```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```

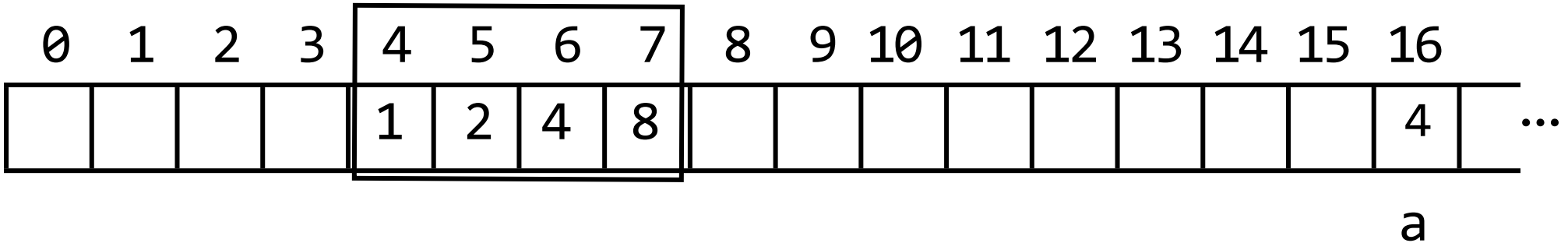




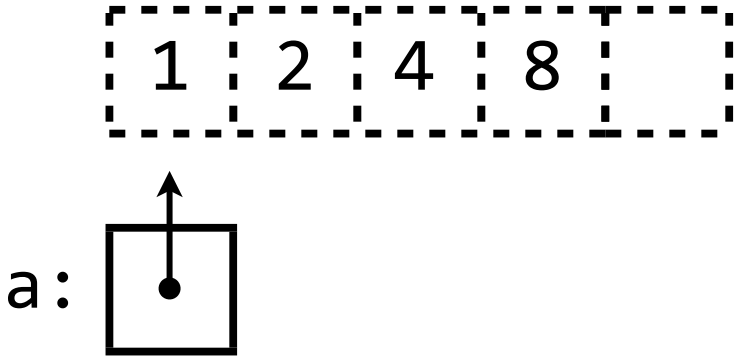
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



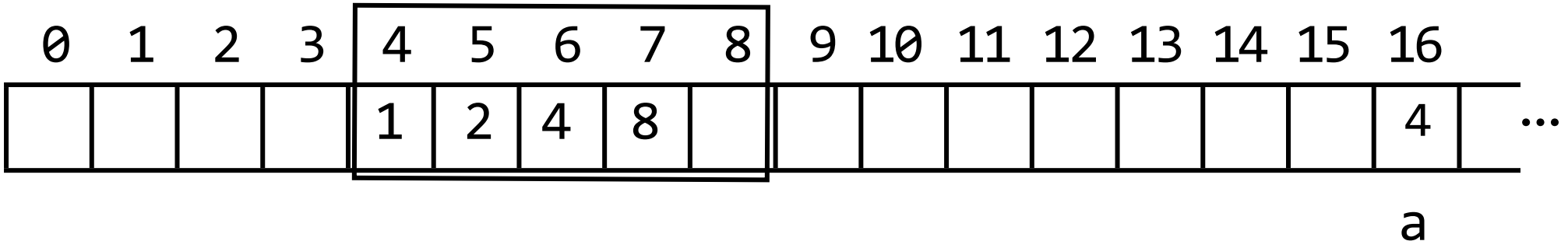

```
a = realloc(a, 5 * sizeof(int));
```



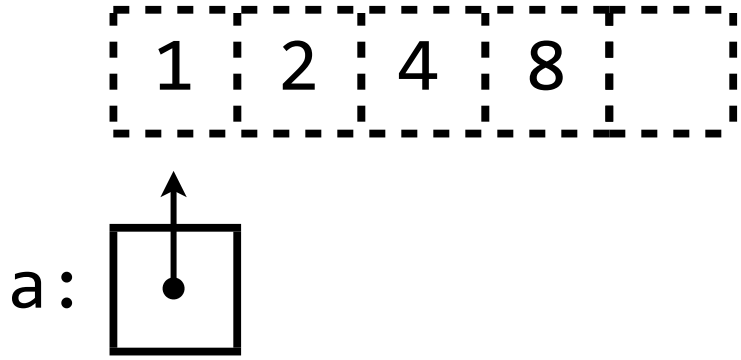
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



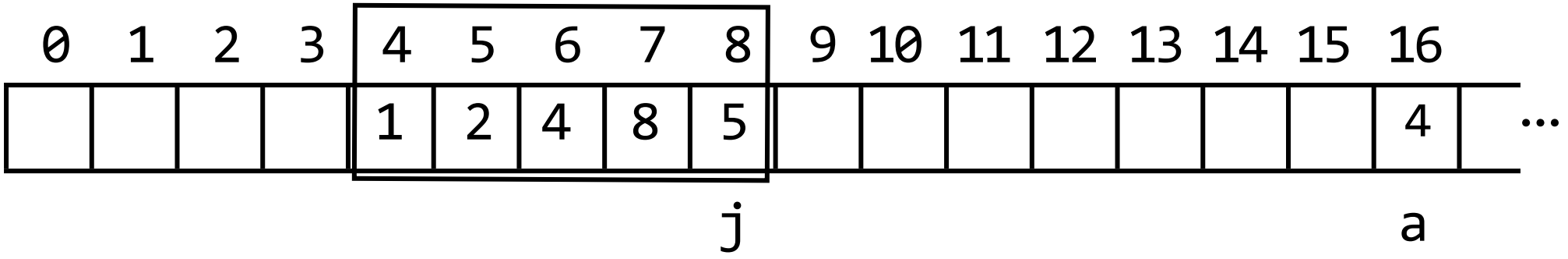

```
a = realloc(a, 5 * sizeof(int));
```



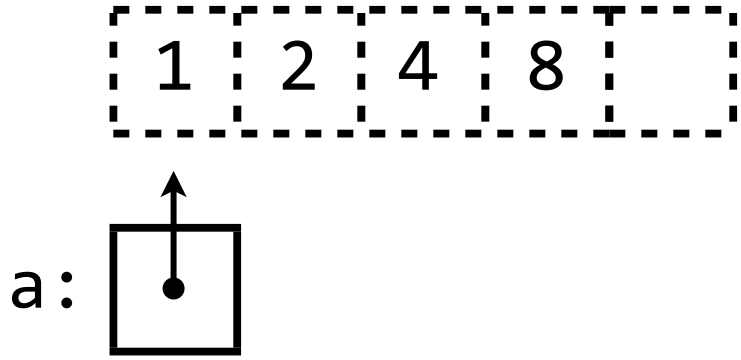
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



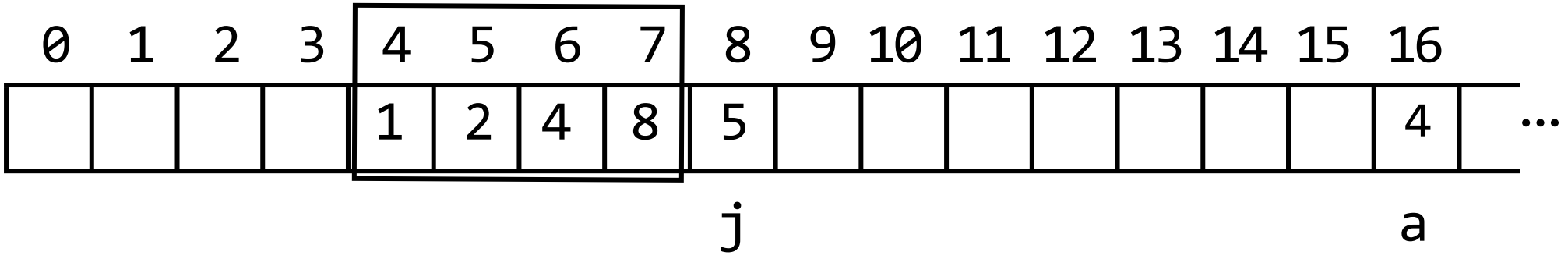

```
a = realloc(a, 5 * sizeof(int));
```



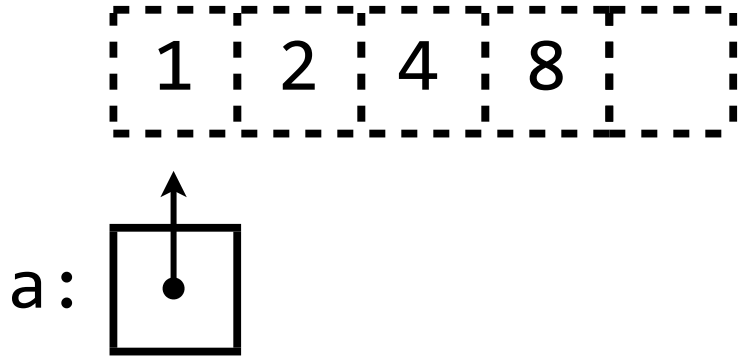
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



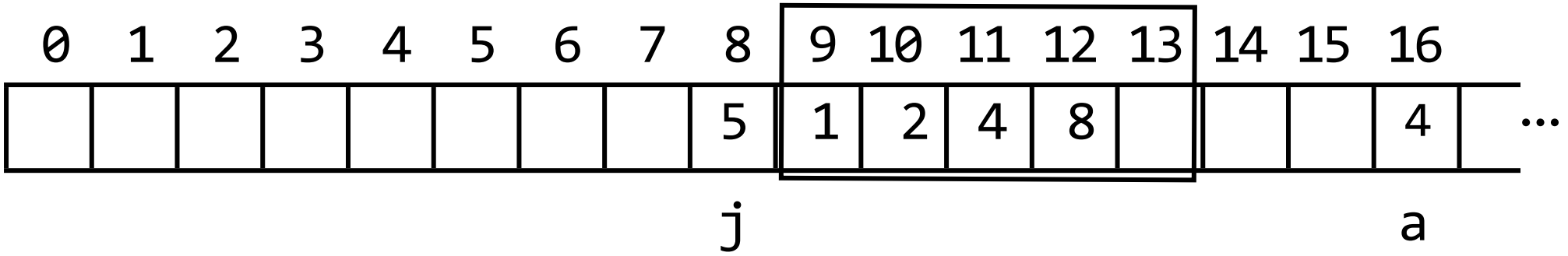

```
a = realloc(a, 5 * sizeof(int));
```



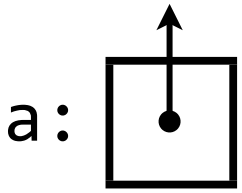
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



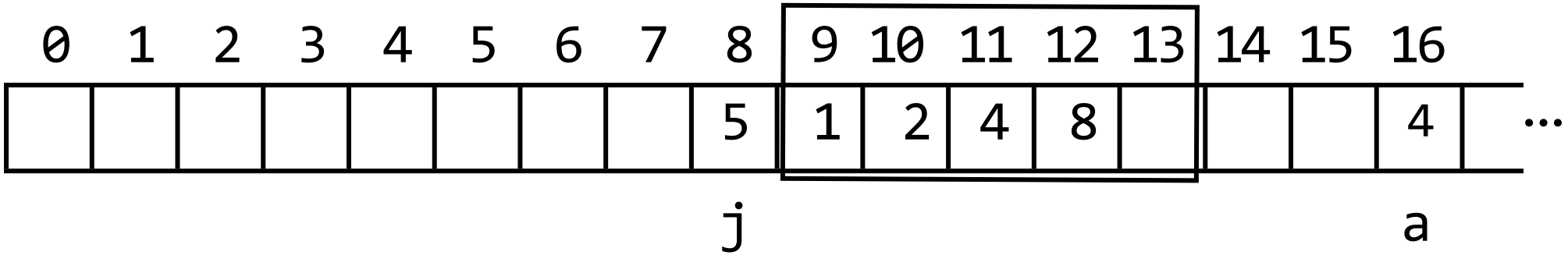

```
a = realloc(a, 5 * sizeof(int));
```



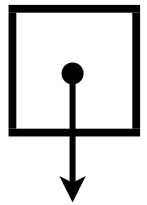
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



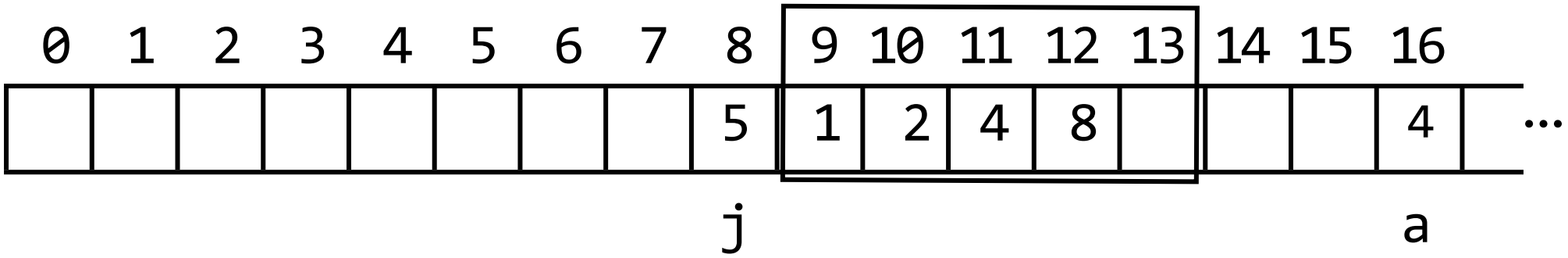

```
a = realloc(a, 5 * sizeof(int));
```



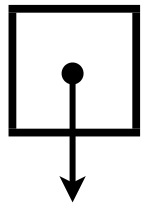
```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```



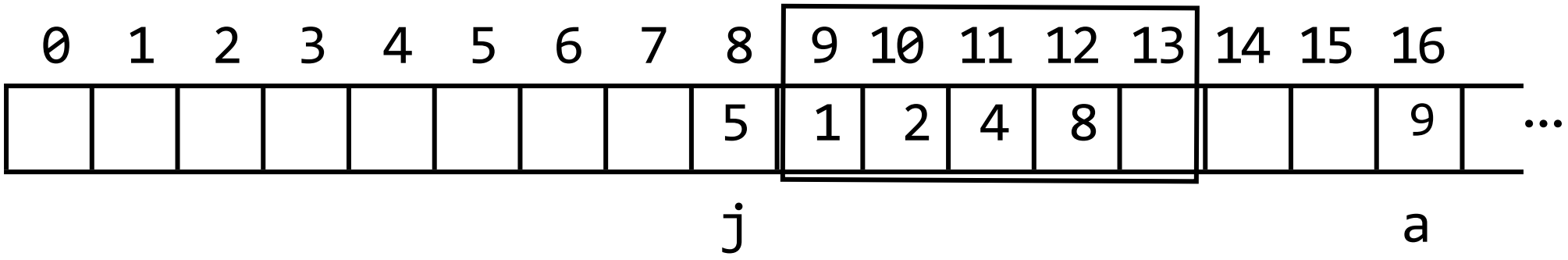

```
a = realloc(a, 5 * sizeof(int));
```



```
int *a = malloc(4 * sizeof(int));
a[0] = 1;
a[1] = 2;
a[2] = 4;
a[3] = 8;
```




```
a = realloc(a, 5 * sizeof(int));
```



Updating Pointers

- After a call to `realloc()` *every* pointer to that chunk of memory must be updated

```
int *largest = &a[1];
...
a = realloc(a, 5 * sizeof(int));
*largest = 10;
```

Wrong

```
largest = &a[1];
*largest = 10;
```

realloc() and NULL

- If realloc() can't resize the memory, it returns NULL
- Technically, you should always check for this
- In this course, we won't

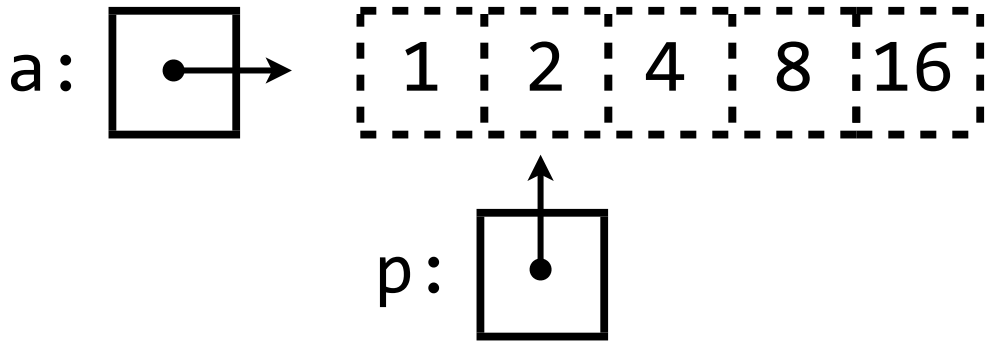
a could be NULL

```
a = realloc(a, 5 * sizeof(int));
```

```
int *resized = realloc(a, 5 * sizeof(int));
if (resized != NULL)
{
  a = resized;
}
else
{
  // handle the error
}
```

Shortening with `realloc()`

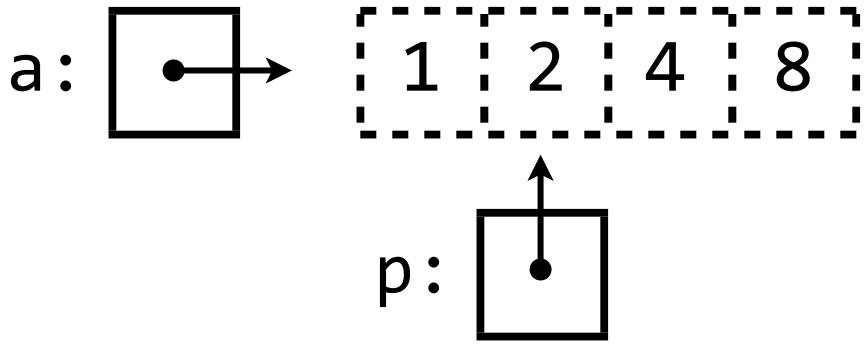
- Shortening an array loses everything that's cut off




```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
```

Shortening with `realloc()`

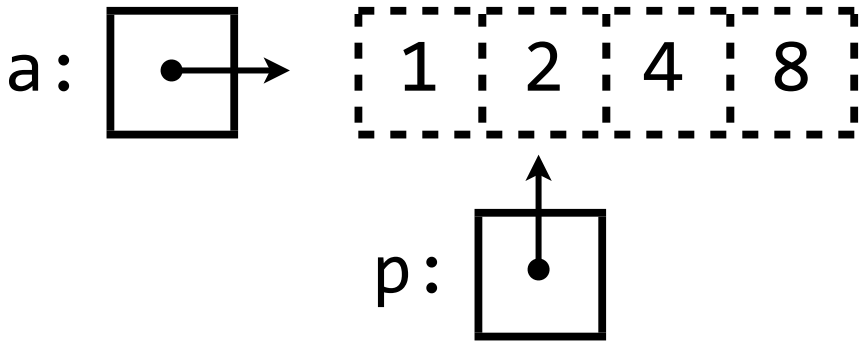
- Shortening an array loses everything that's cut off




```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
a = realloc(a, 4 * sizeof(int));
```

Shortening with `realloc()`

- Shortening an array loses everything that's cut off

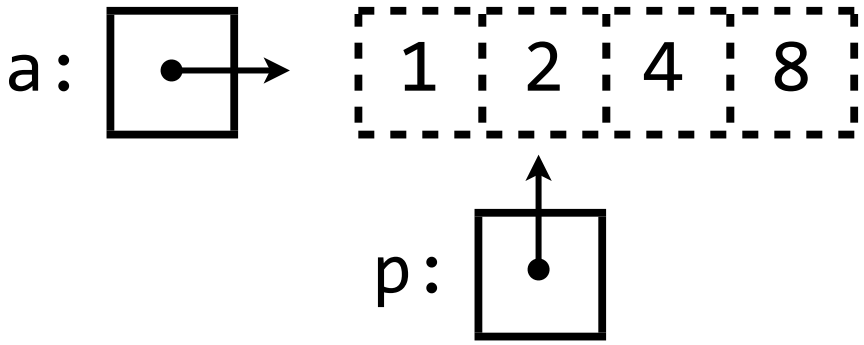



```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
a = realloc(a, 4 * sizeof(int));
a[4] = 32;
```

Wrong

Shortening with `realloc()`

- Shortening an array loses everything that's cut off



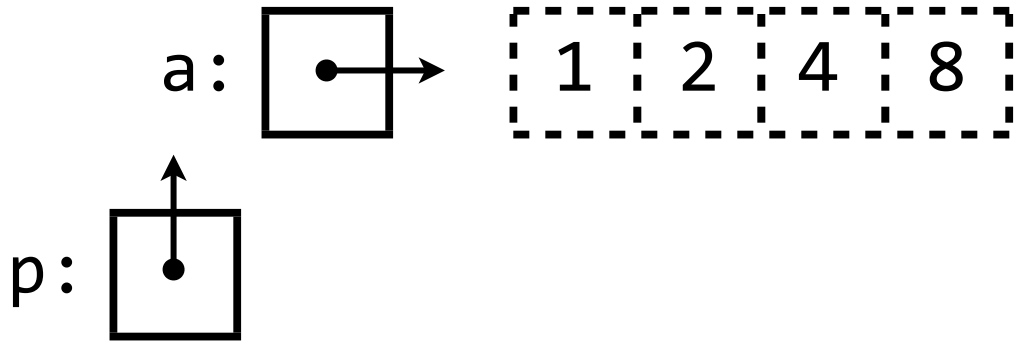

```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
a = realloc(a, 4 * sizeof(int));
a[4] = 32;
*p = 32;
```

Wrong

Wrong

Shortening with `realloc()`

- Shortening an array loses everything that's cut off



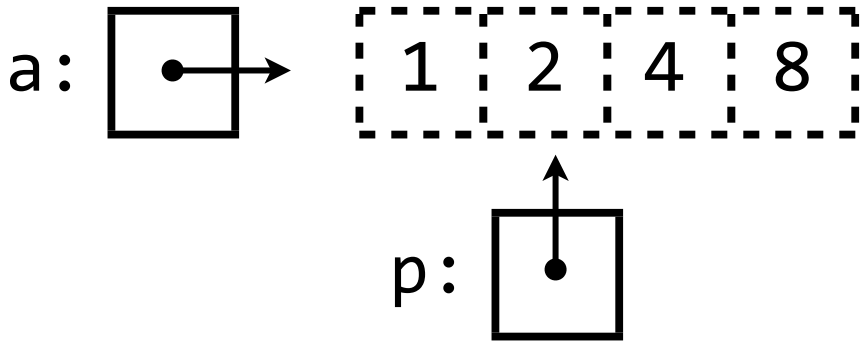

```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
a = realloc(a, 4 * sizeof(int));
a[4] = 32;
*p = 32;
```

Wrong

Wrong

Shortening with `realloc()`

- Shortening an array loses everything that's cut off
- Still need to update pointers




```
int *a = malloc (5 * sizeof(int));
int *p = &a[1];
// Fill in a
a = realloc(a, 4 * sizeof(int));
a[4] = 32;
*p = 32;
 p = &a[1];
```

Wrong

Wrong

`realloc()` Bugs

- Same as `malloc()` and `free()`
  - `realloc()`ing memory you didn't get from `malloc()`
  - Double free by `realloc(p, 0)` after `free(p)`
  - Use after free by `realloc()`ing a `free()`ed pointer
- Forgetting to update pointers

Commenting

Commenting

- Two audiences for your code:
  - Compiler
  - Other people

  - Other people includes future you

Compiler

- Doesn't care about identifier names

```
int iol;
```

```
int indexOfLargest;
```

- Doesn't care about whitespace

```
if (i>0)
{
sum+=1;
}
```

```
if(i>0){sum+=1;}
```

```
if (i > 0)
{
   sum += 1;
}
```

- Doesn't care about comments

```
double angle;
```

```
// Angle of throw, in radians.
double angle;
```

- Cares about syntax

```
double size
int position;
```

```
double size;
int position;
```

~~Compiler~~ Human

- ~~Doesn't~~ Does care about identifier names

```
int iol;
```

```
int indexOfLargest;
```

- ~~Doesn't~~ Does care about whitespace

```
if (i>0)
{
sum+=1;
}
```

```
if(i>0){sum+=1;}
```

```
if (i > 0)
{
   sum += 1;
}
```

- ~~Doesn't~~ Does care about comments

```
double angle;
```

```
// Angle of throw, in radians.
double angle;
```

- ~~Cares~~ Doesn't care as much about syntax

```
double size
int position;
```

```
double size;
int position;
```

```
double t[5];
// (assume t is filled in here)
double ms = 0.0;
int mn = 0;
double fs = 0.0;
int fn = 0;
for (int i = 1; i <= 5; i++)
{
if (t[i] != 0)
{
printf("Time of racer %d is %g\n", i, t[i]);
if (i % 2 == 0)
{
ms += t[i];
mn++;
}
else
{
fs += t[i];
fn++;
}
}
}
double ma = ms / mn;
double fa = fs / fn;
printf("Average times are %g and %g\n", ma, fa);
```

```
double t[5];
// (assume t is filled in here)
double ms = 0.0;
int mn = 0;
double fs = 0.0;
int fn = 0;
for (int i = 1; i <= 5; i++)
{
  if (t[i] != 0)
  {
    printf("Time of racer %d is %g\n", i, t[i]);
    if (i % 2 == 0)
    {
      ms += t[i];
      mn++;
    }
    else
    {
      fs += t[i];
      fn++;
    }
  }
}
double ma = ms / mn;
double fa = fs / fn;
printf("Average times are %g and %g\n", ma, fa);
```

```
double times[5];
// (assume times is filled in here)
double maleSum = 0.0;
int numMaleRacers = 0;
double femaleSum = 0.0;
int numFemaleRacers = 0;
for (int i = 1; i <= 5; i++)
{
  if (times[i] != 0)
  {
    printf("Time of racer %d is %g\n", i, times[i]);
    if (i % 2 == 0)
    {
      maleSum += times[i];
      numMaleRacers++;
    }
    else
    {
      femaleSum += times[i];
      numFemaleRacers++;
    }
  }
}
double maleAvg = maleSum / numMaleRacers;
double femaleAvg = femaleSum / numFemaleRacers;
printf("Average times are %g and %g\n", maleAvg, femaleAvg);
```

```
double times[5];
// (assume times is filled in here)
double maleSum = 0.0;
int numMaleRacers = 0;
double femaleSum = 0.0;
int numFemaleRacers = 0;
for (int i = 1; i <= 5; i++)
{
  if (times[i] != 0)
  {
    printf("Time of racer %d is %g\n", i, times[i]);
    if (i % 2 == 0)
    {
      maleSum += times[i];
      numMaleRacers++;
    }
    else
    {
      femaleSum += times[i];
      numFemaleRacers++;
    }
  }
}
double maleAvg = maleSum / numMaleRacers;
double femaleAvg = femaleSum / numFemaleRacers;
printf("Average times are %g and %g\n", maleAvg, femaleAvg);
```

```
for (int i = 1; i < 5; i++)?

for (int i = 0; i < 5; i++)?

for (int i = 0; i < 4; i++)?
```

```
/* The total number of racers. */
#define NUM_RACERS 4

/* The finishing time (in s) of each racer.
 * The time of racer with bib number n is stored in
 * times[n]. Male racers have even bib numbers, female
 * racers have odd bib numbers. times uses 1-based
 * indexing, since there is no bib number 0.
 * If there is no time recorded for a given bib number,
 * the entry is 0.0. */
double times[NUM_RACERS + 1];

/* The sum of all of the male racers' times. */
double maleSum = 0.0;

/* The number of male racers found in 'times'. */
int numMaleRacers = 0;

/* The sum of all of the female racers' times. */
double femaleSum = 0.0;

/* The number of female racers found in 'times'. */
int numFemaleRacers = 0;
```

```
// Process each element of times, which uses 1-based indexing
for (int i = 1; i <= 5; i++)
{
  if (times[i] != 0)
  {
    printf("Time of racer %d is %g\n", i, times[i]);

    // Even bibs are male, odd bibs are female
    if (i % 2 == 0)
    {
      maleSum += times[i];
      numMaleRacers++;
    }
    else
    {
      femaleSum += times[i];
      numFemaleRacers++;
    }
  }
}
```

```
// Process each element of times, which uses 1-based indexing
for (int i = 1; i <= NUM_RACERS; i++)
{
  if (times[i] != 0)
  {
    printf("Time of racer %d is %g\n", i, times[i]);

    // Even bibs are male, odd bibs are female
    if (i % 2 == 0)
    {
      maleSum += times[i];
      numMaleRacers++;
    }
    else
    {
      femaleSum += times[i];
      numFemaleRacers++;
    }
  }
}
```

```
double t[5];
// (assume t is filled in here)
double ms = 0.0;
int mn = 0;
double fs = 0.0;
int fn = 0;
for (int i = 1; i <= 5; i++)
{
if (t[i] != 0)
{
printf("Time of racer %d is %g\n", i, t[i]);
if (i % 2 == 0)
{
ms += t[i];
mn++;
}
else
{
fs += t[i];
fn++;
}
}
}
double ma = ms / mn;
double fa = fs / fn;
printf("Average times are %g and %g\n", ma, fa);
```

Commenting

- Commenting and style go hand in hand
- Good identifier names helps with commenting

```
/* The sum of all of the male racers' times. */
double maleSum = 0.0;
```

Assumptions

- Assume the reader knows C

```
i++; // Add one to i

// Keep going while sum is less than 0
while (sum < 0)
{
   ...
}

// Create a variable to hold the size
int size;
```

Why, not What

- In general, tell the reader *why* you're doing something, not *what* you're doing
- Anything out of the ordinary

```
// Keep going until we have enough data to process
while (sum < THRESHOLD)
{
   ...
}

// The item's size, in cm
int size;
```

If you were reading this code,
what would you want to know?

Testing

Error Free

- How do we prove that our programs are error free?

Well, we can test them, right?

```
/* Returns the square of 'n' */
int square(int n);
```

```
int n = 2;
printf("%d\n", square(n));
n = 4;
printf("%d\n", square(n));
n = 5;
printf("%d\n", square(n));
n = 52;
printf("%d\n", square(n));
```

4

16

25

2704

Error Free

- How do you prove your programs are error free?

Well, we can test them, right?

- How do you prove that unicorns do not exist?
- How do you prove that this rock repels tigers?

Hmmm...

You Can't Prove a Negative!

- Basic tenet of logic and science
- All you can say is that something is highly unlikely
- More evidence makes it more unlikely

- Since software is essentially math, you can actually do formal proofs on it
- Very difficult and very expensive

```
/* Returns the square of 'n' */
int square(int n);
```

```
int square(int n)
{
  return n * n;
}
```

```
int n = 2;

printf("%d\n", square(n));

n = 4;

printf("%d\n", square(n));

n = 5;

printf("%d\n", square(n));

n = 52;

printf("%d\n", square(n));
```

```
int square(int n)
{
  if (n == 38745)
  {
    int *p = NULL;
    *p = n;
  }

  return n * n;
}
```

Test Cases

- Because we can't test every possible input, we need to select a representative sample
- A good selection is essential

Testing Approaches

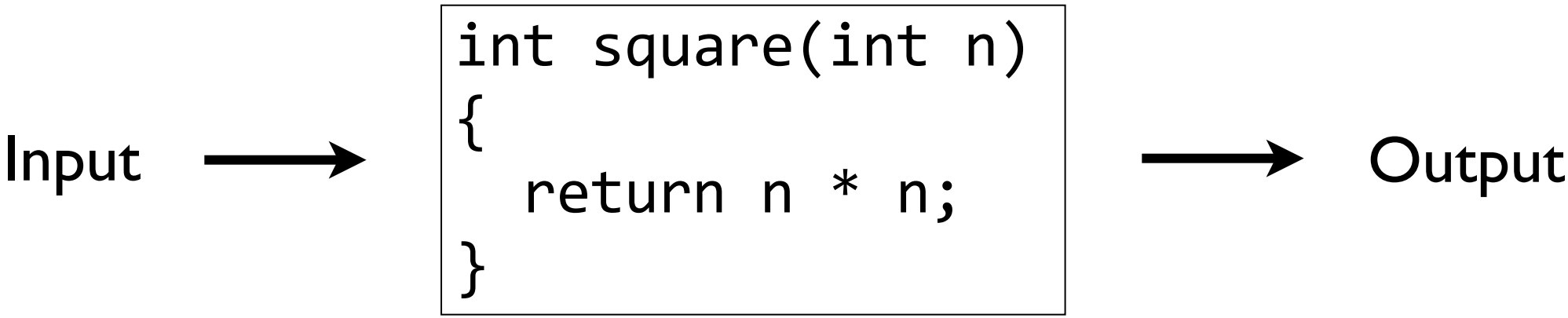
- Black-box testing
- White-box testing

- These are general categories, not exact classifications

Black-box


```
/* Returns the square of 'n' */
```

White-box



Test Cases

- Boundary cases (a.k.a. edge cases)
- Simplest cases
- General cases

```
/* Returns the larger of 'a' and 'b'. */
int max(int a, int b);
```

a	b	*What it's testing*	*Expected*
-2	2	a negative, b positive	2
3	-3	a positive, b negative	3
0	1	0 and 1	1
1	0	0 and 1	1
-1	0	0 and -1	0
0	-1	0 and -1	0
0	0	both 0	?
1	1	both 1	?
65	23	general, a > b	65
564	2193	general, a < b	2193
387	387	general, a == b	?

Special Cases

- When dealing with numbers, consider `0`, `1` (and `-1`?)
  - Other specific cases (e.g., even/odd)
- When dealing with strings, consider empty string, string with one char, multiple null characters (if appropriate)
- When dealing with arrays, consider empty array, array with one item, array with one space, full array
- Need to think about abnormal input, although generally not in this course

Debugging

All Software Has Bugs

- Unavoidable
- Techniques to help avoid them, and to help find them

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

Grace Hopper's Lab Notebook

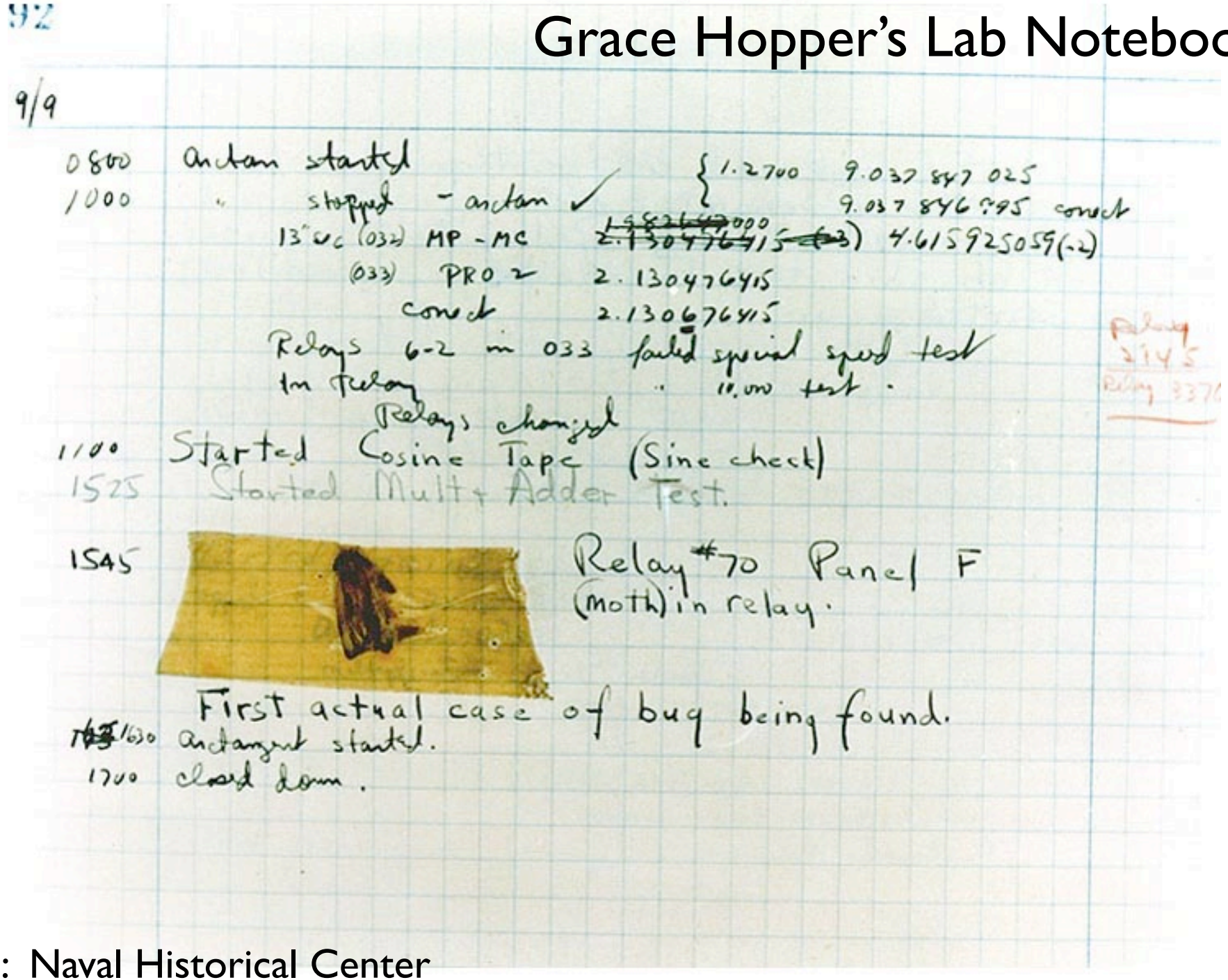


Image: Naval Historical Center

Types of Bugs

- Code that does not do what it's supposed to
- Code that does what it's supposed to, but that isn't the right thing to do
  - May be a bug in the design