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

ECE 220: Computer Systems & Programming

Arrays

Arrays are Groups of Values Named Using an Index

In MP4,
- you have to handle four regions
- with sizes given as **`r1`**, **`r2`**, **`r3`**, and **`r4`**.

In some larger nonograms, a row or column may have 20 regions.

So for MP6 …

… **let's talk about arrays!**

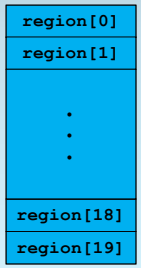
(Instead of **`r1`**, **`r2`**, … , **`r20`**.)

Arrays are Contiguous in Memory; Indices Use Brackets

In **C**, one declares an array of 20 **`int`**s as

```
int region[20];
```

The compiler **allocates memory**
- **to hold 20 `int`s**
- called **`region[0]` through `region[19]`**,
- as illustrated to the right.

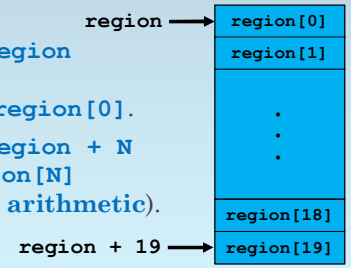


Pointer Arithmetic Moves Among Array Elements

```
int region[20];
```

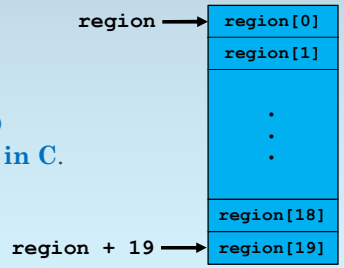
The expression **`region`**
- **has type `int*`**
- and **points to `region[0]`**.

The expression **`region + N`**
- **points to `region[N]`**
- (called **pointer arithmetic**).



Brackets are Shorthand for Add and Dereference

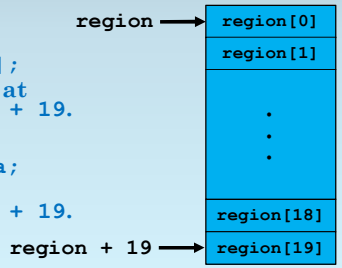
```
int region[20];
```



Thus,
- `region[N]` and
- `*(region + N)`
- are **equivalent in C**.

Use Brackets for Reading and Writing Array Elements

```
int region[20];
int a;
```



Thus
- **`a = region[19];`**
- **reads the value at address `region + 19`.**

And
- **`region[19] = a;`**
- **stores bits to address `region + 19`.**

Pointer Arithmetic Generally Involves Multiplication

```
int region[20];
```

Say that **`region` is address `0x12345000`**.

What is `region + 5`?

`0x12345005`? Not necessarily.

The answer depends on
- the size of an **`int`** and
- the addressability of memory.

The amount added is the number of addresses required for 5 `int`s.

Pass Array Arguments as Pointers

Let's do an **example**. Our task:
- write **a subroutine**
- to **find the minimum value**
- **in an array of `int32_t`s**.

How can we pass the array to the subroutine?

Copy it onto the stack?

Expensive!

Instead, **pass a pointer to the first element!**

An Address Does Not Define an Array Length

```
int32_t min_value
    (int32_t const values[]);
```

same as `int32_t const* values`

Look good?

How can `min_value` know the array size?

As shown, it cannot.

So …?

Add a second parameter for the length.

Finding Minimum Value with a C Function

```
int32_t min_value
    (int32_t const values[], int32_t n_values)
{
    int32_t min = values[0];
    int32_t check;
    for (check=1; n_values > check; check++) {
        if (min > values[check]) {
            min = values[check];
        }
    }
    return min;
}
```

Assume first value is smallest.

In loop body, **`check`** goes from **`1`** to **`n_values - 1`**.

Finding Minimum Value with a C Function

```
int32_t min_value
    (int32_t const values[], int32_t n_values)
{
    int32_t min = values[0];
    int32_t check;
    for (check=1; n_values > check; check++) {
        if (min > values[check]) {
            min = values[check];
        }
    }
    return min;
}
```

If smaller value found, copy value to **`min`**.

Return smallest value found.

Using Our Minimum Value Function

How do we use the function?

```
int32_t my_nums[4] =
    {93, 100, 79, 42};
int32_t least;

least = min_value (my_nums, 4);
```

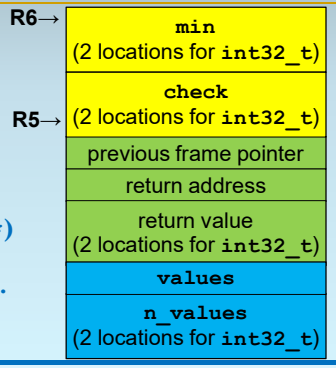
Initializes array to values shown.

pointer to **`my_nums[0]`**

Holds 42 after assignment.

length of my_nums

LC-3 Stack Frame for `min_value`



Note that values (type `int const*`) takes only one memory location.

Strings Can Be Stored in Arrays of `char`s

Strings are like (and can be stored in) **arrays of `char`s.**

Unfortunately, we **must choose a size** for an array.

```
char name[20];
printf ("Hi, what is your name?");
if (1 == scanf ("%s", name)) {
    printf ("Hello, %s!\n", name);
}
```


Array Bounds are Not Checked in C

```
char name[20];
printf ("Hi, what is your name?");
if (1 == scanf ("%s", name)) {
    printf ("Hello, %s!\n", name);
}
```

What happens if human user types more?

Hopefully, the program crashes…

* *

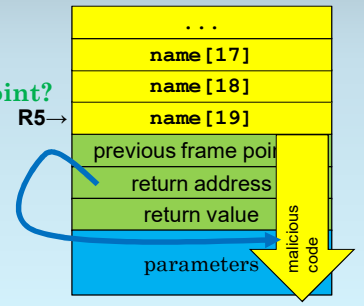
LC-3 Stack Frame for Current Function

Typed "name" can overwrite return address.

Where does it point?

Anywhere the user wants!

Including into the code they just wrote into your machine.



Buffer Overrun Attacks Used to Dominate Vulnerabilities

This type of attack is
- a **buffer overrun attack**,
- the dominant software vulnerability
- for many years.

Microsoft went through 50 million lines of code to try to eliminate them.

Recent OS changes have also helped:
- reduced ability to execute code on stack, and
- randomization of code location.

Use Field Width to Make `scanf` Safe

```
char name[20];
printf ("Hi, what is your name?");
if (1 == scanf ("%19s", name)) {
    printf ("Hello, %s!\n", name);
}
```

Use field width 19* to limit input to 19 characters (need 1 char for NUL).

*Solutions (such as this one) that require humans to maintain them are error-prone.

Impromptu Survey on Phone Books

How many of you have …

- **…used a phone book?**
- **…seen a phone book?**
- **…heard of phone books?**

Just wondered.

How Do We Search When Values are Sorted?

Imagine that you have
- an array of integers
- sorted in numerically increasing order.

How do you check whether
- **a particular integer**
- **appears in the array?**

Let's write a **C** function and return either
- the index of the desired value, or
- -1 if the value is not in the array.

Parameters and Local Variables for Binary Search

```
int32_t binary_search
    (int32_t array[], int32_t len,
     int32_t value)
{
    int32_t low = 0;
    int32_t high = len - 1;
    int32_t mid;
```

length of array

a sorted array of integers

number to find in array

initialize search bounds [low,high] to [0,len – 1]

Main Iteration: Look Once, Then Adjust Bounds

Keep looking until we have no place to look.

```
    while (high >= low) {
        mid = low + (high - low) / 2;

        // look at one value
        // and adjust bounds
    }
    return -1;
}
```

Look in the middle … but why this way?

Value not found: return -1.

Bugs Can Be Subtle and Hide for Decades

Are these two expressions equivalent?

```
low + (high - low) / 2
(low + high) / 2
```

No …

- but for 20+ years,
- library code for binary search
- used the second expression.

Sums Can Overflow, Producing Negative Array Indices

```
low + (high - low) / 2
(low + high) / 2
```

Consider the following:

- $0 \le$ `low` $< 2^{31}$ and $0 \le$ `high` $< 2^{31}$, and
- `low` $\le$ `high`, so $0 \le$ `(high - low)` $< 2^{31}$ *

What about `low + high`?

Overflow can produce `mid < 0` !

*Technically, one needs to couple this argument with a proof by induction.

Loop Body: Find Value, or Adjust One Bound

```
while (high >= low) {
    mid = low + (high - low) / 2;
    if (value == array[mid]) {
        return mid;  // Found!
    }
    if (value < array[mid]) {
        high = mid - 1;
    } else {
        low = mid + 1;
    }
}
```


Can Use Arrays of Arrays for Multidimensional Data

One can create arrays of arrays. For example,

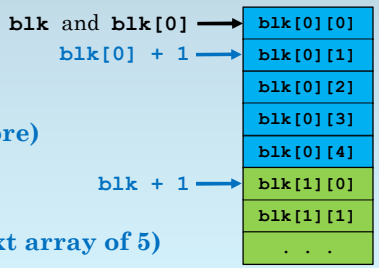
```
int32_t blk[3][5];
```

allocates an array of 3 arrays of 5 `int32_t`s.

The expressions **`blk` and `blk[0]`**
- **have the same value**
- but **`blk[0]` has type `int32_t*`**
- while **`blk` has type `int32_t (*)[5]`**
- (pointer to an array of 5 `int32_t`s)

Remember: Pointer Arithmetic Depends on Size

What happens when we add 1…



… to blk[0]?
(same as before)

…to blk?
(points to next array of 5)

Often Need to Map Multidimensional Data by Hand

You will use multidimensional arrays in MP6.

In later MPs,
- array dimensions are not known in advance,
- so your program must
- perform the mapping into 1D.

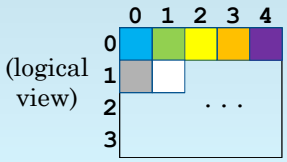
For example, in MP8,
- an **`image`** is **`height × width`** pixels,
- but **`height`** and **`width`** are variables.

Multiply "Larger" Dimension by Size of Smaller

We choose which dimension to map first.

Here, row index **y** is multiplied by **width**:

(x,y) → y * width + x



(logical view)

(layout in memory)

Let's Have Some Fun

Time for some fun…

Let's play cards!

You can teach me …

… how to sort my hand.

Parameters and Local Variables for Insertion Sort

```
void insertion_sort
(int32_t values[], int32_t num_vals)
{
    int32_t sorted, current, index;
```

the cards (each 0 to 12)

the number of cards

number sorted (after loop body)

card to insert

position to insert

Main Loop: Sort One Card at a Time Until All are Sorted

One card is always sorted, so start with 2.

Done when **num_vals** cards are sorted.

```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
    // insert one more card
}
```

Sort one card at a time.

loop invariant: before loop body, first (**sorted – 1**) cards are sorted

Loop Body: Find the Place to Insert the New Card

New card is at position (**sorted – 1**).

```
current = values[sorted - 1];
// find place to insert new card
values[index] = current;
```

After inner loop, **index** is correct position for new card.

Loop Over Possible Positions Until Correct One is Found

Look first at new card's original position.

```
for (index = sorted - 1; 0 < index;
     index--) {
    // check possible position
    values[index] =
        values[index - 1];
}
```

Position cannot be < 0.

Check one position per iteration.

Move card over if new card is smaller.

Position Check Simply Compares Card Values

```
if (current >= values[index - 1]) {
    break;
}
```

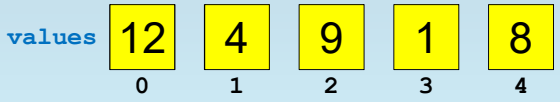
If the new card's value is at least as great as the card in the previous position, stop searching.

Reference Copy of Insertion Sort

```
void insertion_sort (int32_t values[], int32_t num_vals)
{
    int32_t sorted, current, index;
    for (sorted = 2; num_vals >= sorted; sorted++) {
        current = values[sorted - 1];
        for (index = sorted - 1; 0 < index; index--) {
            if (current >= values[index - 1]) {
                break;
            }
            values[index] = values[index - 1];
        }
        values[index] = current;
    }
}
```


Let's Try It! Start with Some Arguments

```
void insertion_sort
(int32_t values[], int32_t num_vals)
```

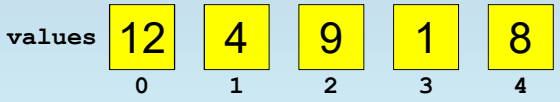


num_vals is 5

Our Local Variables are Not Initialized

```
int32_t sorted, current, index;
```

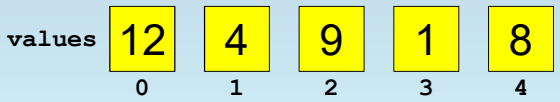
What's in these variables?
Bits!



num_vals is 5

First Loop Iteration (First Card is Sorted Already)

```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
```



Is num_vals >= sorted? Yes!

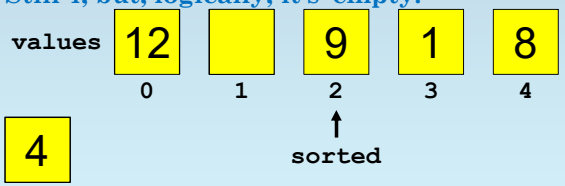
sorted

num_vals is 5

Time to Insert the 4 at the Right Position

```
current = values[sorted - 1];
```

Huh? What's in values[1]?
Still 4, but, logically, it's 'empty.'

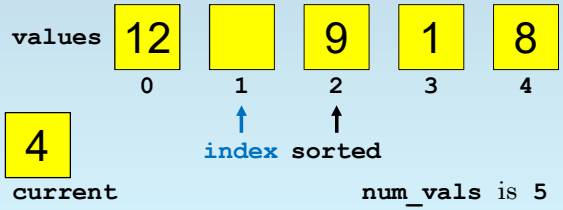


num_vals is 5

Find the Right Place for the 4

```
for (index = sorted - 1; 0 < index;
     index--) {
```

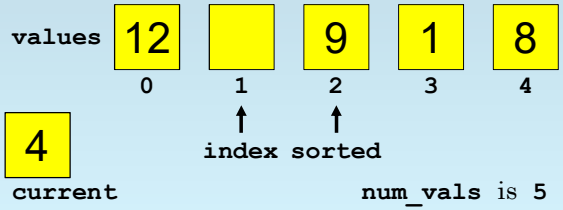
Is `0 < index`? Yes!



Compare 4 with 12

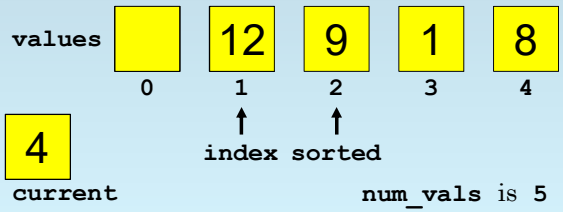
```
if (current >= values[index - 1]) {
    break;
}
```

Is `4 >= 12`? No.



Copy 12 from Position 0 to Position 1

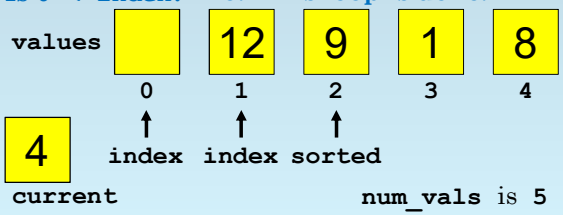
```
values[index] = values[index - 1];
```



Update and Test Again

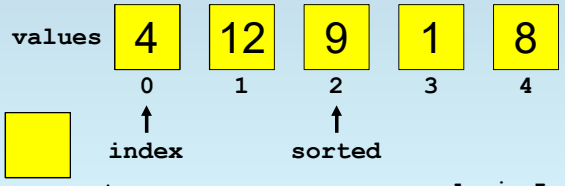
```
for (index = sorted - 1; 0 < index;
     index--) {
```

Is `0 < index`? No. This loop is done.



Place New Card in "Blank" Position (**index**)

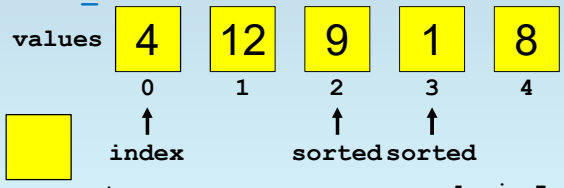
```
values[index] = current;
```



Update and Test Again (First 2 Cards Now Sorted)

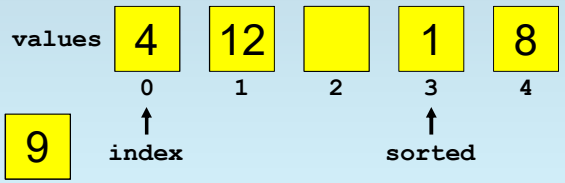
```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
```

Is `num_vals >= sorted`? Yes.



Time to Insert the 9 at the Right Position

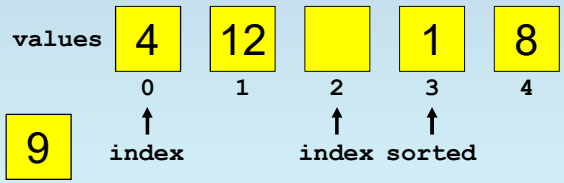
```
current = values[sorted - 1];
```



Find the Right Place for the 9

```
for (index = sorted - 1; 0 < index;
     index--) {
```

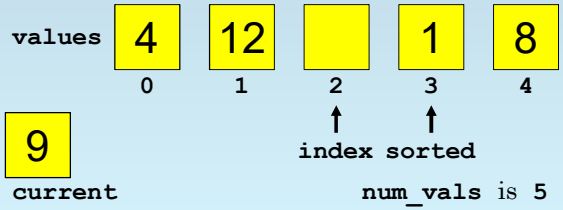
Is `0 < index`? Yes!



Compare 9 with 12

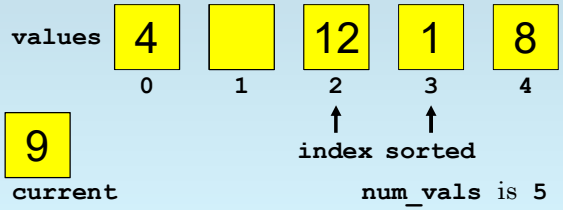
```
if (current >= values[index - 1]) {
    break;
}
```

Is 9 >= 12? No.



Copy 12 from Position 1 to Position 2

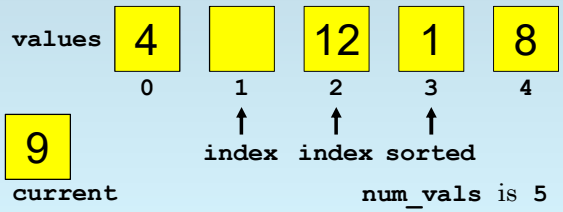
```
values[index] = values[index - 1];
```



Update and Test Again

```
for (index = sorted - 1; 0 < index;
     index--) {
```

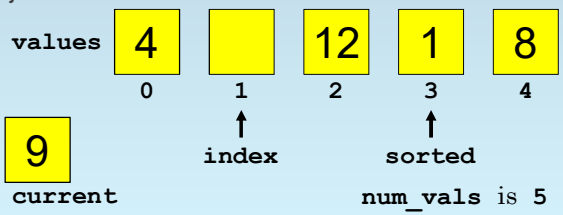
Is 0 < index? Yes.



Compare 9 with 4

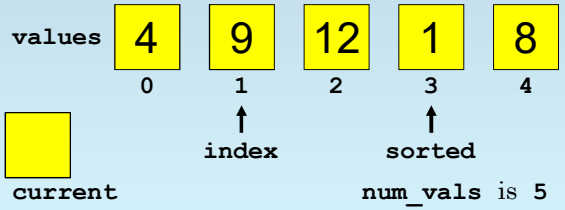
```
if (current >= values[index - 1]) {
    break;
}
```

Is 9 >= 4? Yes, so break.



Place New Card in "Blank" Position (**index**)

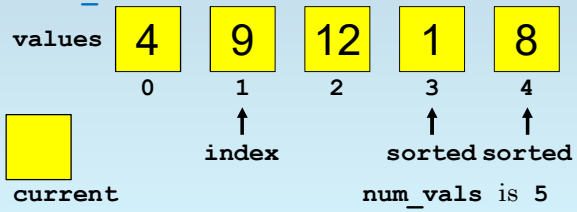
```
values[index] = current;
```



Update and Test Again (First 3 Cards Now Sorted)

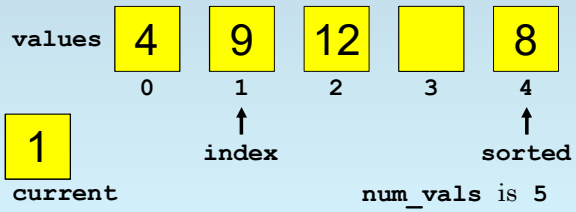
```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
```

Is `num_vals >= sorted`? Yes.



Time to Insert the 1 at the Right Position

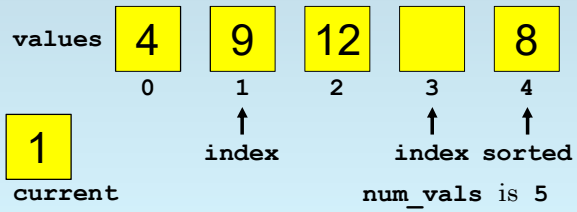
```
current = values[sorted - 1];
```



Find the Right Place for the 1

```
for (index = sorted - 1; 0 < index;
     index--) {
```

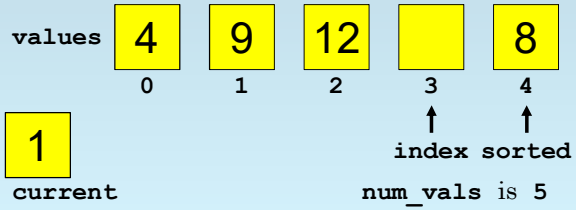
Is `0 < index`? Yes!



Compare 1 with 12

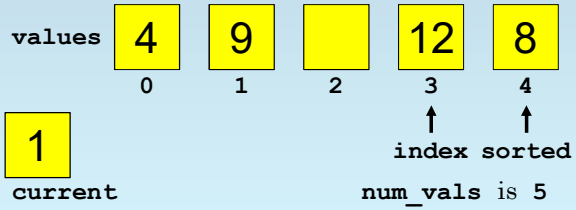
```
if (current >= values[index - 1]) {
    break;
}
```

Is 1 >= 12? No.



Copy 12 from Position 2 to Position 3

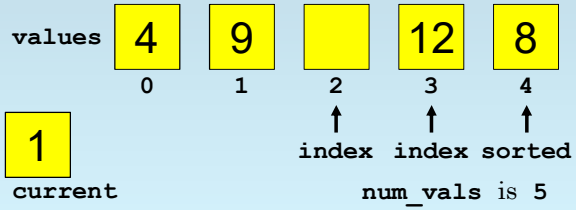
```
values[index] = values[index - 1];
```



Update and Test Again

```
for (index = sorted - 1; 0 < index;
     index--) {
```

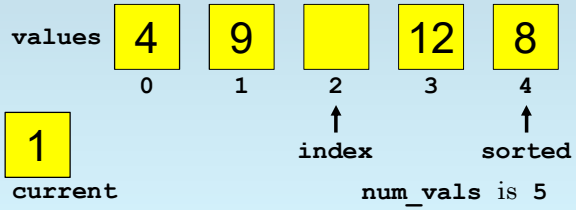
Is 0 < index? Yes.



Compare 1 with 9

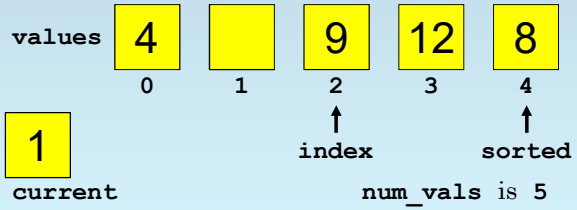
```
if (current >= values[index - 1]) {
    break;
}
```

Is 1 >= 9? No.



Copy 9 from Position 1 to Position 2

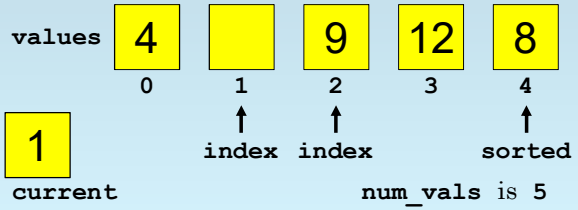
```
values[index] = values[index - 1];
```



Update and Test Again

```
for (index = sorted - 1; 0 < index;
     index--) {
```

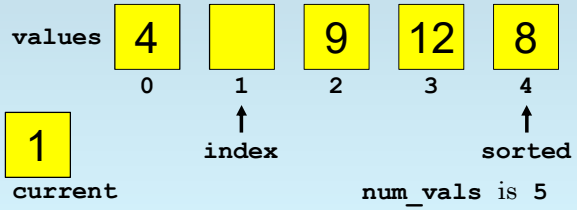
Is `0 < index`? Yes.



Compare 1 with 4

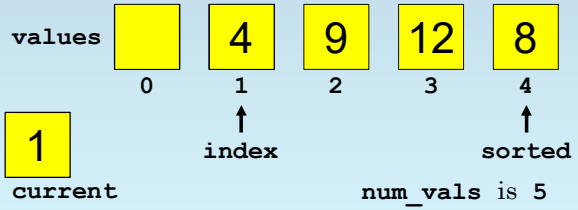
```
if (current >= values[index - 1]) {
    break;
}
```

Is `1 >= 4`? No.



Copy 4 from Position 0 to Position 1

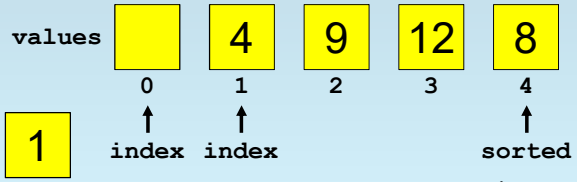
```
values[index] = values[index - 1];
```



Update and Test Again

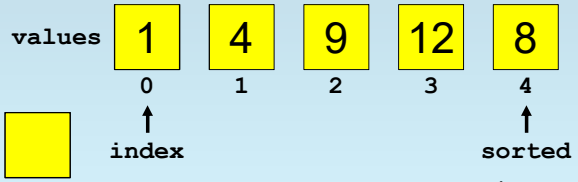
```
for (index = sorted - 1; 0 < index;
     index--) {
```

Is `0 < index`? No. This loop is done.



Place New Card in "Blank" Position (`index`)

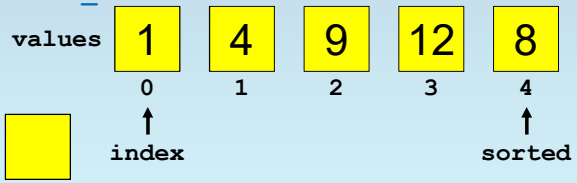
```
values[index] = current;
```



Update and Test Again (First 4 Cards Now Sorted)

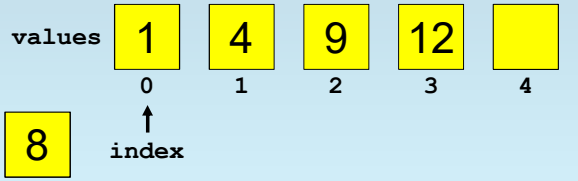
```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
```

Is `num_vals >= sorted`? Yes.



Time to Insert the 8 at the Right Position

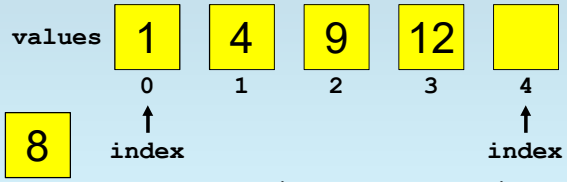
```
current = values[sorted - 1];
```



Find the Right Place for the 8

```
for (index = sorted - 1; 0 < index;
     index--) {
```

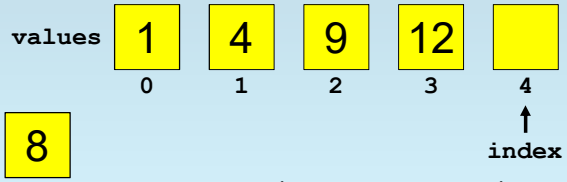
Is `0 < index`? Yes!



Compare 8 with 12

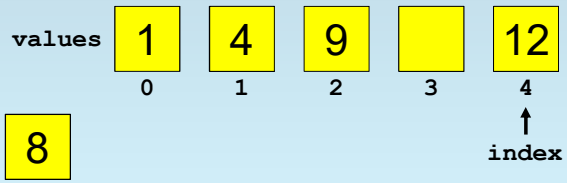
```
if (current >= values[index - 1]) {
    break;
}
```

Is `8 >= 12`? No.



Copy 12 from Position 3 to Position 4

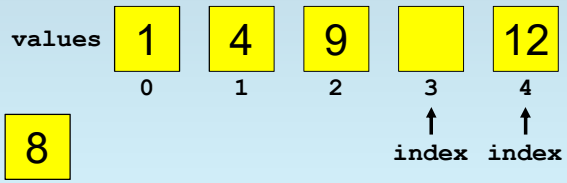
```
values[index] = values[index - 1];
```



Update and Test Again

```
for (index = sorted - 1; 0 < index;
     index--) {
```

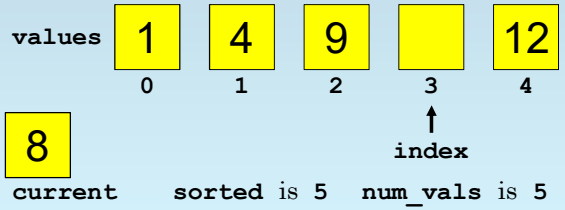
Is `0 < index`? Yes.



Compare 8 with 9

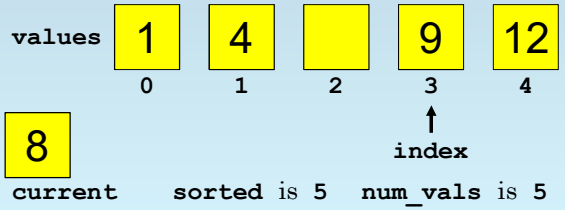
```
if (current >= values[index - 1]) {
    break;
}
```

Is 8 >= 9? No.



Copy 9 from Position 2 to Position 3

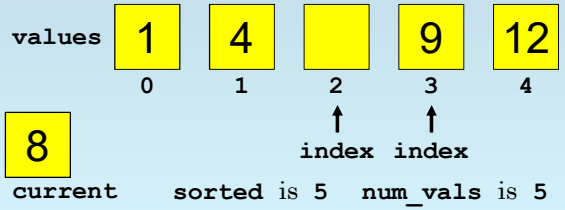
```
values[index] = values[index - 1];
```



Update and Test Again

```
for (index = sorted - 1; 0 < index;
     index--) {
```

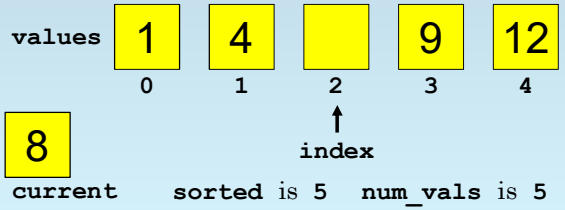
Is 0 < index? Yes.



Compare 8 with 4

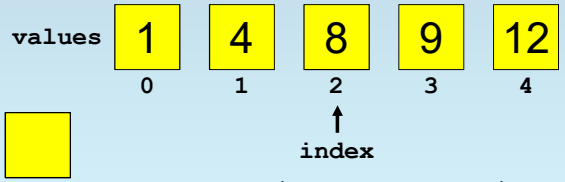
```
if (current >= values[index - 1]) {
    break;
}
```

Is 8 >= 4? Yes, so break.



Place New Card in "Blank" Position (**index**)

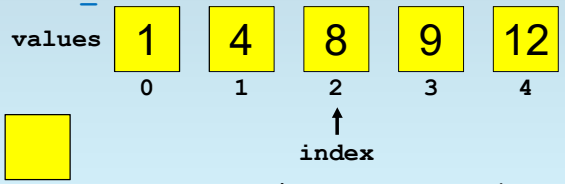
```
values[index] = current;
```



Update and Test Again (First 5 Cards Now Sorted)

```
for (sorted = 2; num_vals >= sorted;
     sorted++) {
```

Is num_vals >= sorted? No. We're done.



Time for Another Think-Pair-Share

As before, let's do a group exercise in lecture.

The process:

1. I give you a problem.
2. You form groups of 3-4 people.
3. Talk about ways to solve the problem.
4. Once enough of the groups have finished, one group volunteers to share their answer.
5. We go over the group's answer together.

The Task: Calculate Average of a List of Numbers

The task…

- Allow a user to enter up to 5 numbers (not -1).
- End the list by typing -1 (not include in list).
- Find and print average of user's numbers.

One half of the class writes **main**.

The other half writes **calc_avg**.

Let's use

```
int32_t calc_avg
  (int32_t const array[], int32_t len);
```

