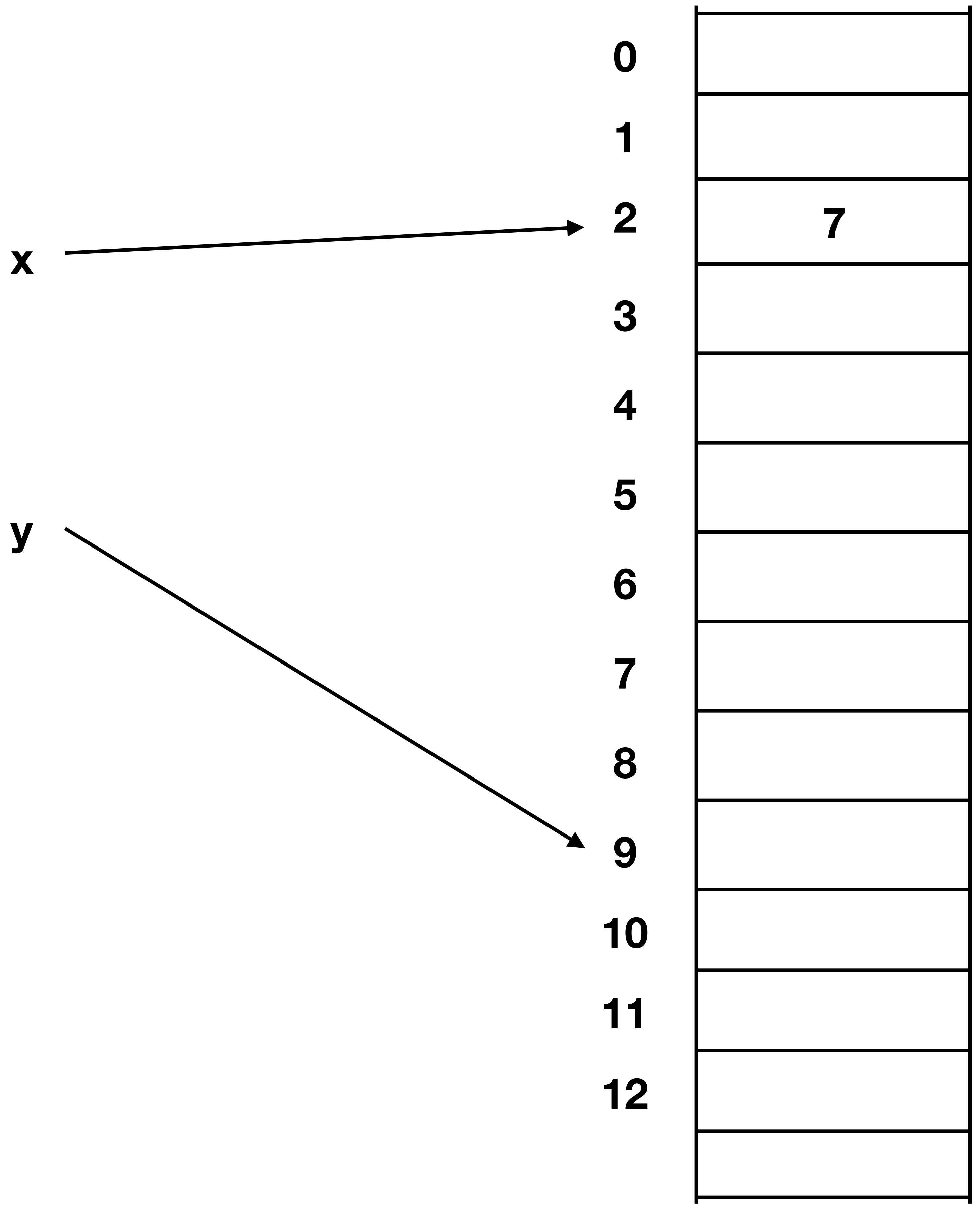
Lecture 8

Some pointers

Variables

- A variable has a type (e.g. int, char etc..)
- It has a name (eg. int x, int y)
- It can have a value
- It has an address in the memory of your computer.


x
y
0
1
2
3
4
5
6
7
8
9
10
11
12
7

& and * operators

Suppose x is an integer variable.

& is an operator which is applied to the name of a variable and it gives the actual address of where that variable is stored in memory.

The address of x in this example is memory location 2, so in the example on the last slide, &x would give us 2, similarly &y would give us 9.

* is an operator which is applied to an address in memory and gives us the value stored at that address. So in the example *&x gives us 7.

So we can get the address where the value of x is stored,
how do we display the value stored in the variable x

There are 2 ways we can do so.

```
printf(“%d”, x) ;
```

Or

```
printf(“%d”, *&x) ;
```

The * is an operator that expects us to give it an address and it
will then give us the value stored at that address. Because we know the address
of the variable y we can display the contents

```
printf(“%d”, *9) ;
```

Consider the following statements

```
    int x ;
    int y ;
    x = 7 ;
    y = *&x;

    printf("The address of the variable x is %p\n", &x ) ;

    printf("The value of the variable x using the name is %d\n", x )

    printf("The value of the variable x using *&x is %d\n", *&x ) ;

    printf("The value of y is %d\n", y) ;
```

This is what was displayed when I ran it.

```
The address of the variable x is 0x7ffeeac20ae4
The value of the variable x using the name is 7
The value of the variable x using *&x is 7
The value of y is 7
```

A variable which points to an integer (i.e. stores the address of that integer) is called a pointer.

We can declare a variable to point to an integer

```
int *p ;
```

We can now give it a value

```
p = &x ;
```

We can change the value at the address &x by doing this

```
*p = *p + 1 ;
```

Now the value of p has not changed but the value at x has.

```
If we have the following
        int x ;
        int *y ;
        x = 7 ;
        y = &x ;

        printf("x= %d", x) ;
        printf(" and y = %p \n", y) ;
```

We get this output

```
x= 7 and y = 0x7ffee20c8ae4
```

I we add a few more lines

```
    printf("x= %d", x) ;
    printf(" and y = %p \n", y) ;

    *y = *y + 1 ;

    printf("x= %d", x) ;
    printf(" and y = %p \n", y) ;
```

We get the following output

```
x= 7 and y = 0x7ffee20c8ae4
x= 8 and y = 0x7ffee20c8ae4
```

x has changed but y hasn’t

Now we can think of doing some things that could be a little dangerous.

We could write functions that had “side effects”.

That means that the function didn’t just return a value to the main program when it was finished, it could also change the value of variables in the main program.

This might be useful but please be very careful when you try it.

Let us look at an example.

```
void swap (int *x, int *y)
{
     int temp ;
     temp = *x ;
     *x = *y ;
     *y = temp ;
}
```

This is a function that expects to be given the addresses of 2 integer variables. It will swap the values stored at those locations.

```
If we execute these statements

      int x ;
      int y ;
      x = 7 ;
      y = 4 ;

      printf("x = %d and y =  %d\n", x, y) ;
      swap (&x, &y);
      printf("x = %d and y =  %d\n", x, y) ;
```

Remember swap expects to get the addresses of 2 integer variables, so we need &

This is what is displayed

```
x = 7 and y =  4
x = 4 and y =  7

But we didn't change x and y in the main function.
```

When you read a value into an integer
using scanf you give scanf the address of the
integer variable

```
int x ;

scanf(“%d”, &x ) ;

// x now has a value and this is a “side effect” of the function scanf
```

Arrays in C.

When you declare an integer array in C

```
int f [10] ;
```

f is actually a pointer to the address of f[0]

When you pass an array as a parameter to a function in C you are passing in the address of the first element in that array.

The following function expects to get the name of a 1-dimensional array and a variable indicating the size of the array.

```
int a_sum ( int a [], int size )
{
        int i ;
        int sum ;

        i = 0 ;
        sum = 0 ;

        while (i != size)
        {
                sum = sum + a[i] ;
                i = i+1 ;
        }
        return sum ;
}
```

```
int main()
{
        int f [] = {1,2,2,3,4,5,6,7,8,9} ;

        printf(" the a_sum of f is %d \n", a_sum (f, 10));
}
```

Here we call the function a_sum and provide it with the name of the array and the size of the array.

Running this gives the following

```
 the a_sum of f is 47
```

This function expects to get an int pointer and a size.
We know the int pointer points to the start of an array,
so that is why we use a[i] in the body of the function.

```
int p_sum ( int *a , int size)
{
        int i ;
        int sum ;

        i = 0 ;
        sum = 0 ;

        while (i != size)
        {
                sum = sum + a[i] ;
                i = i+1 ;
        }
        return sum ;
}
```

```
int main()
{
        int f [] = {1,2,2,3,4,5,6,7,8,9} ;

        printf(" the p_sum of f is %d \n", p_sum (f, 10));
}
```

Here we call the function p_sum and provide it with the name of the array and the size of the array.

Running this gives the following

```
 the p_sum of f is 47
```

Here the function expects to get an int pointer and a size
However, neither the parameter declaration or the way in which
*a is used within the function explicitly tell us that we are
summing the values in an array.

```
int pp_sum ( int *a, int size )
{
        int i ;
        int sum ;

        i = 0 ;
        sum = 0 ;

        while (i != size)
        {
                sum = sum + *(a+i) ;
                i = i+1 ;
        }
        return sum ;
}
```

```
int main()
{
        int f [] = {1,2,2,3,4,5,6,7,8,9} ;

        printf(" the pp_sum of f is %d \n", pp_sum (&f[0], 10));
}
```

Here we call the function pp_sum and provide it with the address of the first element in the array and the size of the array.

Running this gives the following

```
 the pp_sum of f is 47
```

When we write programs we should always try to make them
as clear and simple and readable as possible.

So, don’t try to make the program difficult to understand.

Suppose we have an array int f [10]
which already has values in it.

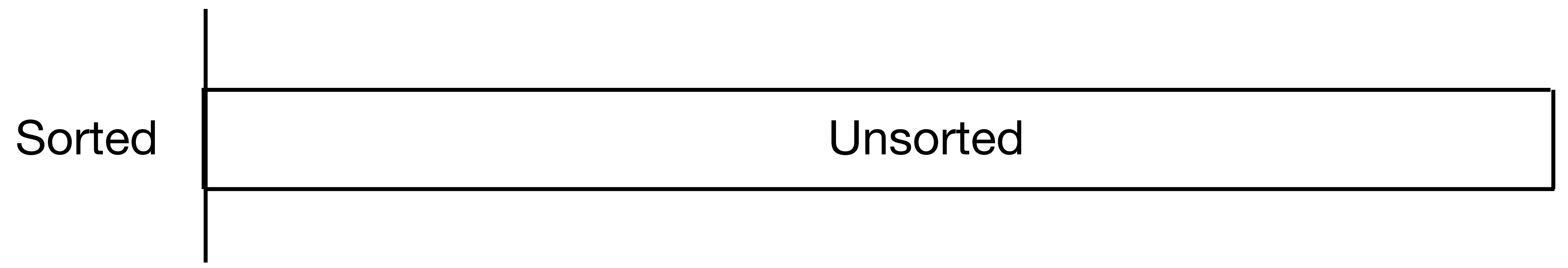
We would like to rearrange those values so that they were ascending.

This is called sorting the array into ascending order.

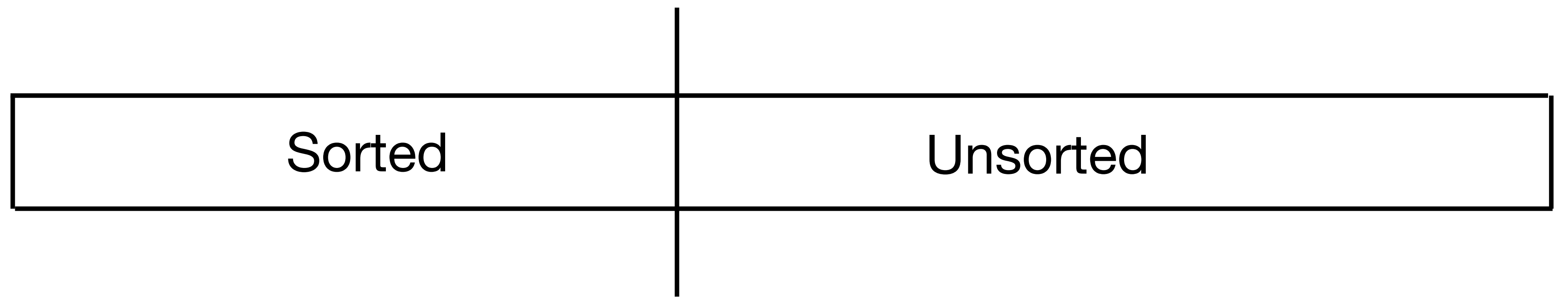
There are many different algorithms that can be used for sorting.

Today we look at 2 of them.

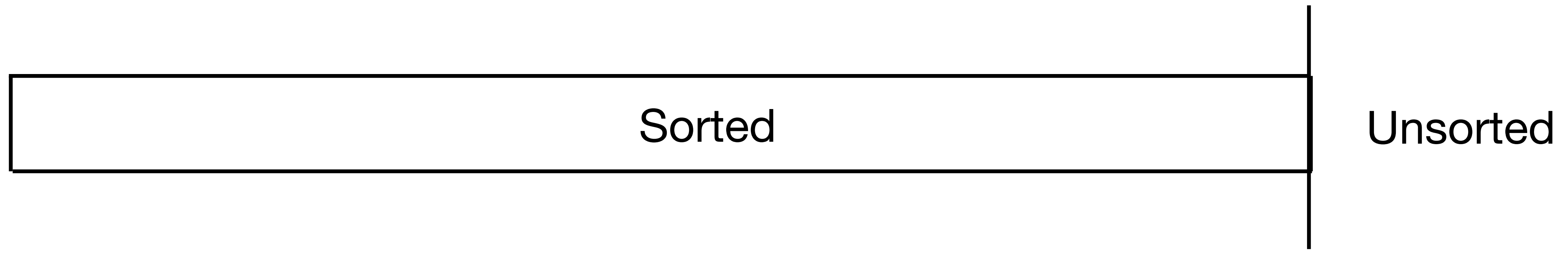
Before we start (Sorted part is empty)



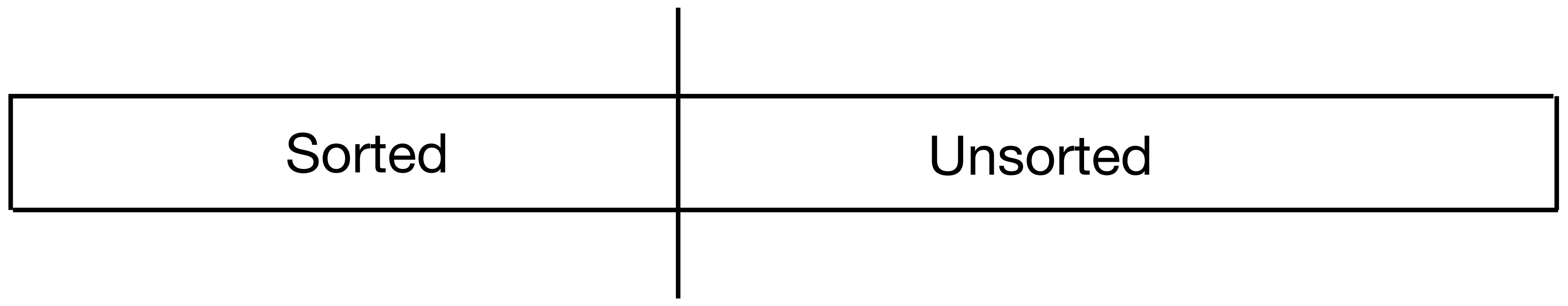
During the sorting (Some sorted and some unsorted)



When we finish (Unsorted part is empty)



To make progress we need to move the line to the right.

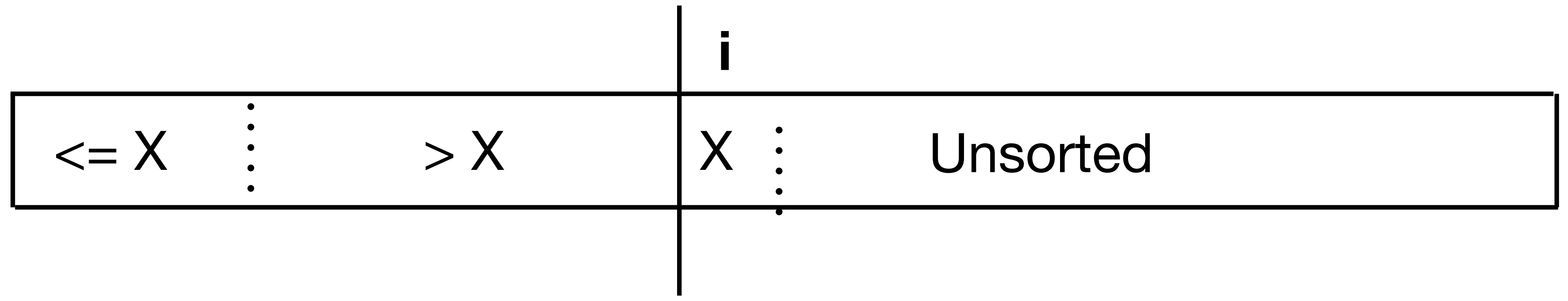


How should we do this?

Everything to the left of the line is sorted and we are currently looking at f[i]. Let is call the value in there X.

In the Sorted part of the array we can imagine it is divided into 2 parts those sorted values which are <= X and those sorted values that are > X.

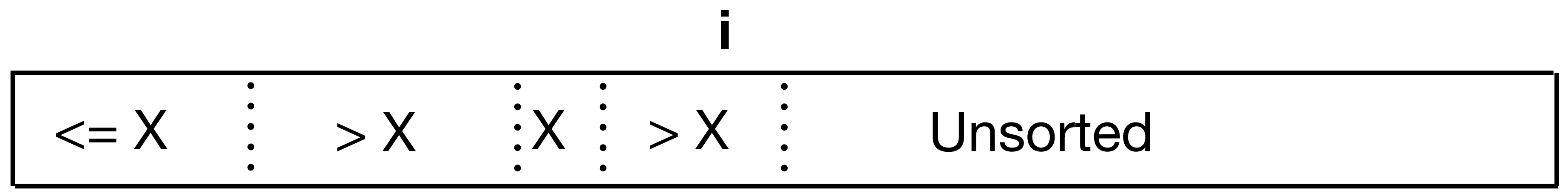
Remember that either of those could be empty.



We would like to insert the value at f[i] into the correct place so that everything up to and including f[i] is now sorted.

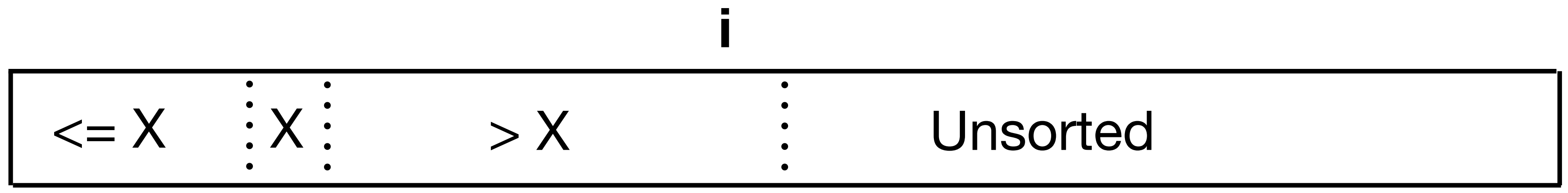
```
i = 0 ;
while (i != 10)
{
  “Insert f[i] into correct place so
   that f[0..i] is sorted”
  i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

During the process we might have a picture like this



It seems that the value X is moving down through the group of values that are greater than it.

We will finish when X finally reaches the start of the section which are <=X



```
We use a function which takes the addresses of 2 ints
and swaps the values at those addresses.

void swap (int *x, int *y)
{
        int temp ;
        temp = *x ;
        *x = *y ;
        *y = temp;
}
```

```
int i ;
int j ;

i = 0 ;
while (i != 10)
{
   j = i ;
   while (( j != 0 ) && ( f[j] < f[j-1] ))
   {
      swap( &f[j], &f[j-1] ) ;
      j = j-1;
   }
   i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

Note.

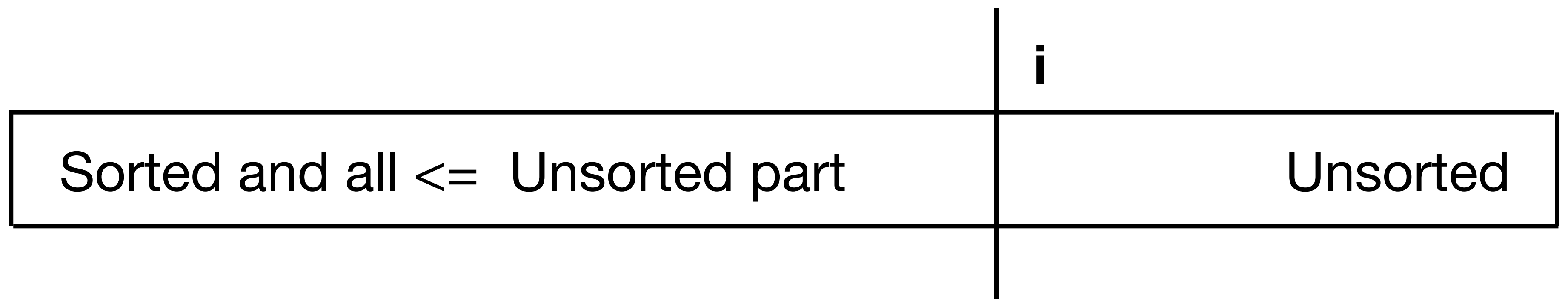
```
   while (( j != 0 ) && ( f[j] < f[j-1] ))
```

When j == 0 we hope that we never get to
test f[j] < f[j-1] because that would be out of bounds.

In C if we have C1 && C2
Then C2 is only tested if C1 is true.

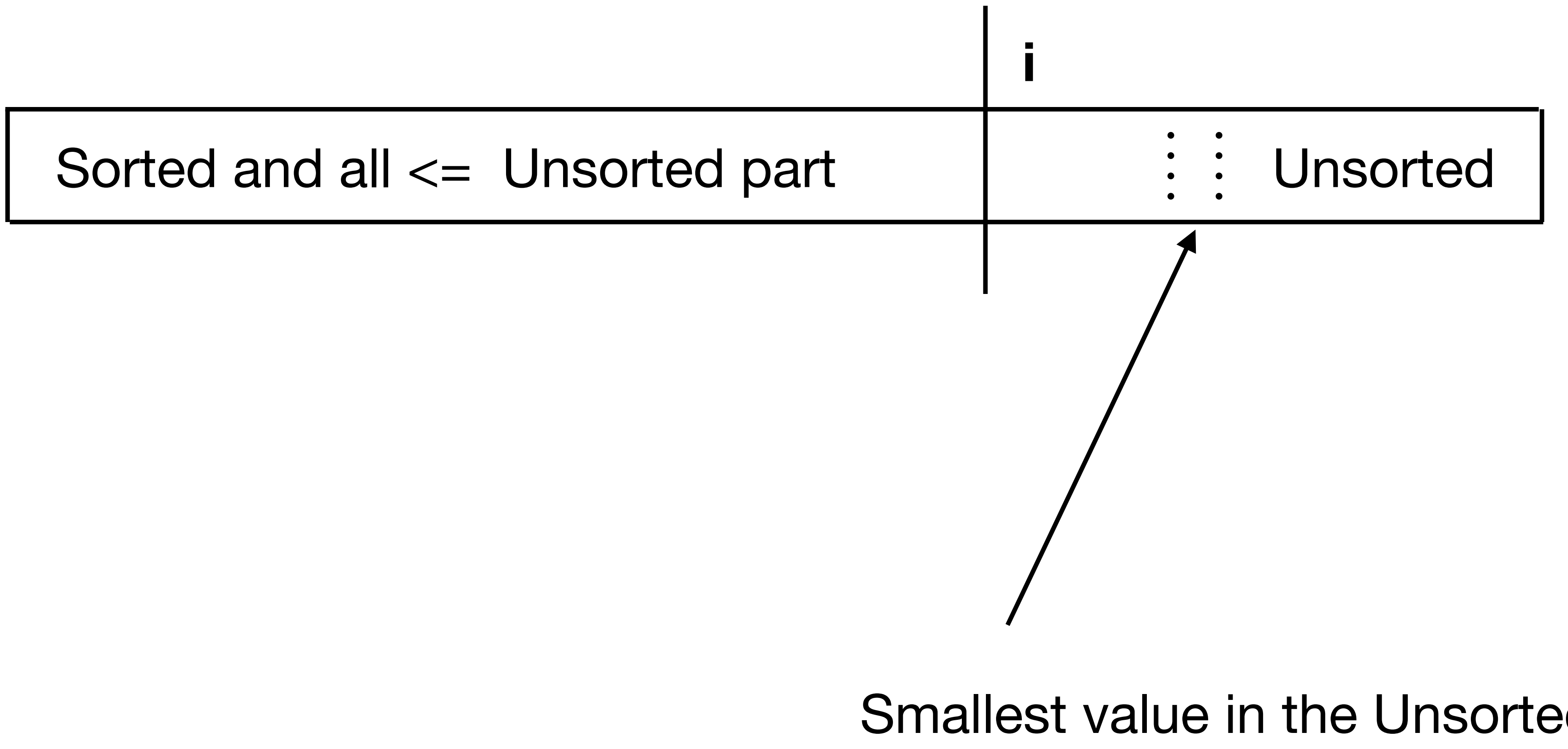
Be careful as not all programming languages evaluate boolean expressions
in this way.

Let us go back a few slides. We were looking at how to extend the sorted section of the array so as to make progress and move the line to the right.



Let us change our picture a little and make the part to the left of the line all sorted but also all <= to the values in the Unsorted part

Now to make progress we need to find the smallest value in the Unsorted part of the array and swap it with the value in f[i]. Then we can extend the Sorted part to include f[i]


i
Sorted and all <= Unsorted part
Unsorted
Smallest value in the Unsorted part

```
i = 0 ;
while (i != 10)
{
   “Find the smallest values in the unsorted part
   and swap it with the value in f[i]”
   i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

```
int i ;
int j ;
int min_index ;
i = 0 ;
while (i != 10)
{
 j = i ;
 min_index = i ;
 while ( j!= 10)
 {
     if (f[j] <= f[min_index])
         { min_index = j ; }
     j = j+1 ;
 }
// f[min_index] is smallest value in unsorted part

 swap(&f[i], &f[min_index]) ;
 i = i+1 ;

}

// everything up to position i is sorted and i == 10
```

Insertion Sort

```
int i ;
int j ;
int min_index ;
i = 0 ;
while (i != 10)
{
 j = i ;
 min_index = i ;
 while ( j!= 10)
 {
     if (f[j] <= f[min_index])
         { min_index = j ; }
     j = j+1 ;
 }
 swap(&f[i], &f[min_index]) ;
 i = i+1 ;
}
// everything up to position i is sorted and i == 10
```

Selection sort

```
int i ;
int j ;

i = 0 ;
while (i != 10)
{
   j = i ;
   while (( j != 0 ) && ( f[j] < f[j-1] ))
   {
      swap( &f[j], &f[j-1] ) ;
      j = j-1;
   }
   i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

Sorting and speed

Suppose your array contains N elements

If you use either of the Sorting programs we have looked at then the outer loop will be performed N times.

For each of these, the inner loop will be performed on average N/2 times.

So the time it takes to run the program will be proportional to N * (N/2)

We usually write this as O(N*N)

This is called the Time Complexity or the Temporal Complexity of the program

As N gets bigger N*N gets bigger faster.

Sorting and speed

N	N^2	N^3	
10	100	1000	
100	10000	1000000	
1000	10^6	10^9	
10000	10^8	10^12	

If a computer can perform 1,000,000 instructions per second then a program which has 10^12 instructions would need 1,000,000 seconds to run, that is 16,666 minutes, or 277 hours, or about 5 and a half days.

Complexity and speed

The best ways to improve the speed of a program are to try to redesign it so as to go from O(N^3) to O(N^2) or from O(N^2) to O(N).

In future modules we will study some techniques to allow you to do this.

Saving a few iterations in a loop really doesn’t make much of a difference, so don’t try too hard to improve efficiency that way.