Lecture 5

Reasoning about loops

Assignment 1

- You can now submit your assignment on Moodle.
- Please submit the program before midnight tomorrow.

Suppose we are asked to write a program to compute the product of the first 10 strictly positive natural numbers.

The natural numbers start at 0

The strictly positive natural numbers start at 1

Here is a list of the numbers

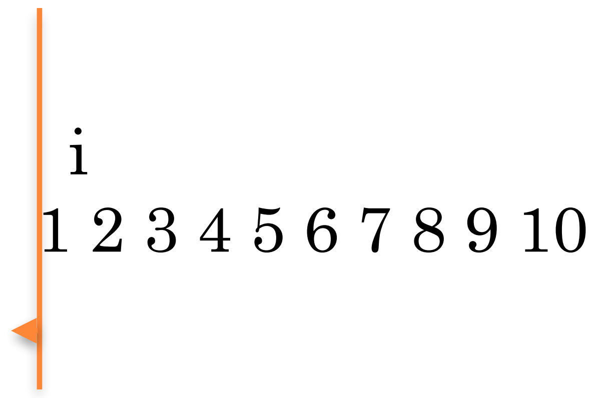
    1 2 3 4 5 6 7 8 9 10

I am going to make use of 2 integer variables

    p will store the product of the values
    i will store the current value

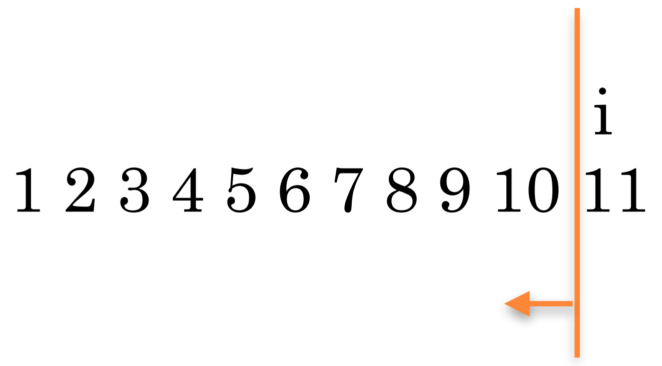
now let us study this problem

Here is a picture at **the start**.



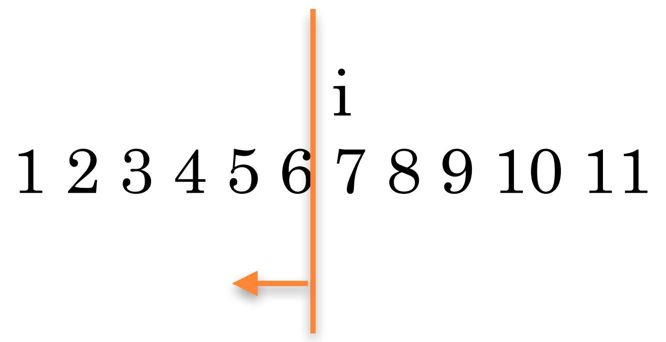
everything to the left of the line has been multiplied and stored in variable p.

Here is a picture at **the end**.



everything to the left of the line has been multiplied
and stored in variable p

Here is a picture at **some point in between**.

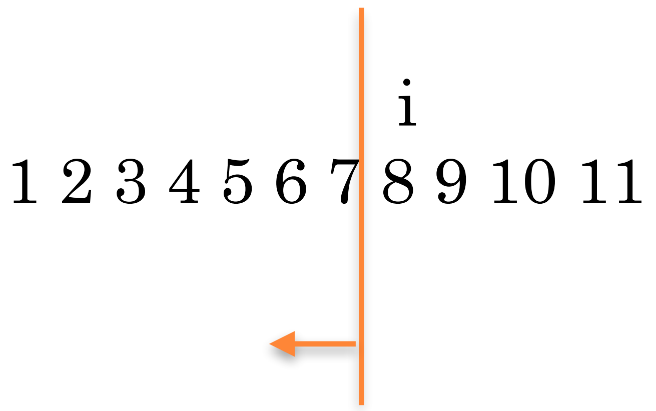


everything to the left of the line has been multiplied
and stored in variable p

Now what do you think you should do here to make progress?

Multiply p by the current value 7
Move the line one place to the right.

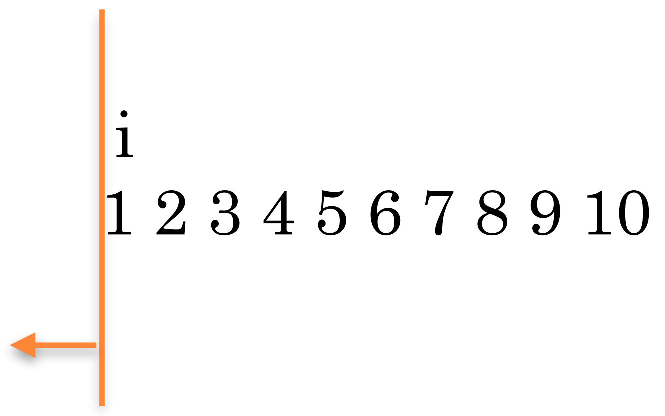
Here is a picture at **some point in between**.



everything to the left of the line has been multiplied
and stored in variable p

We have multiplied p by the current value 7 and
moved the line one place to the right.

Here is a picture at **the start**.



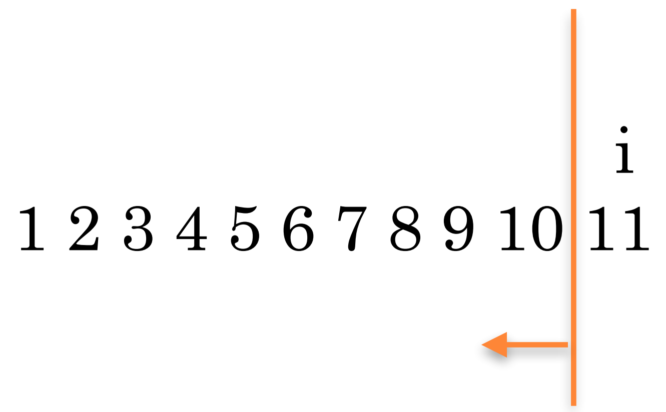
everything to the left of the line has been multiplied
and stored in variable p

Expressed in C

```
p = 1 ;
i = 1 ;
```

Make sure you set p to 1, remember you are multiplying.

Here is a picture at **the end**.

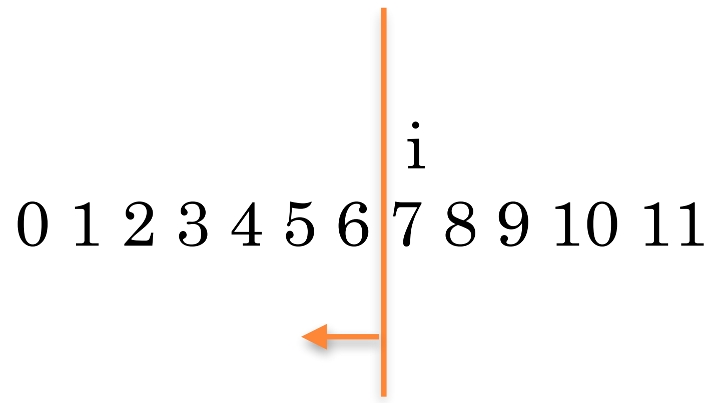


everything to the left of the line has been multiplied and stored in variable p

At this point we are finished so i == 11
So when we are not finished the guard on the loop will be i != 11

```
while ( i != 11)
        {
        }
```

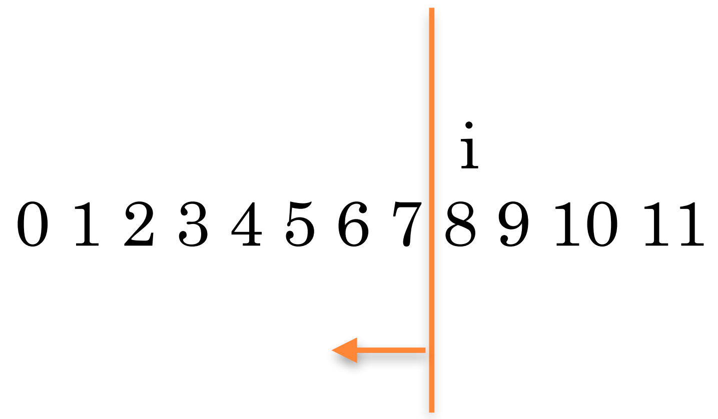
Here is a picture at **some point in between**.



everything to the left of the line has been multiplied and stored in variable p

We now need to multiply by the next value and then move the line on by 1.

Here is a picture at **some point in between**.



everything to the left of the line has been multiplied and stored in variable p

We now multiply by the next value and then move the line on by 1.

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

Finished program

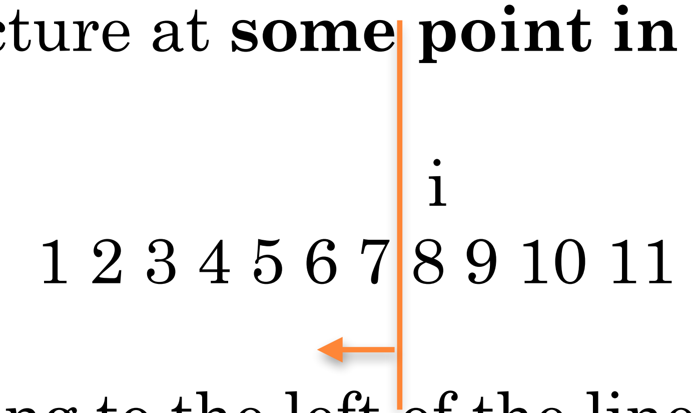
```
#include <stdio.h>
main ()
    {
    long int p ;
    int i ;

    p = 1 ;
    i = 1 ;
    while ( i != 11)
        {
        p = p *  i ;
        i = i + 1 ;
        }
    printf ( “the product of the first 10 pos naturals : %ld \n”, p) ;
    }
```

I want you to study the 2 examples we just did.

Ask yourself…

> How are they similar?
>
> How are they different?

Here is a picture at **some point in between**.



“everything to the left of the line has been multiplied and stored in variable p”

The picture and text above are describing something which is true at different times in the program.

They should be true before the loop starts.
They should be true after the loop finishes.

In the body of the loop we want to keep them true but to move the line to the right.

The text and picture describe something
which is called the **Loop Invariant.**

Invariants play a major role in allowing us to reason about
and build programs.

In later stages you will learn a lot more about them and how to use them.
For now, I would like you to draw the pictures to help you when you are writing
a loop.

Review: arrays

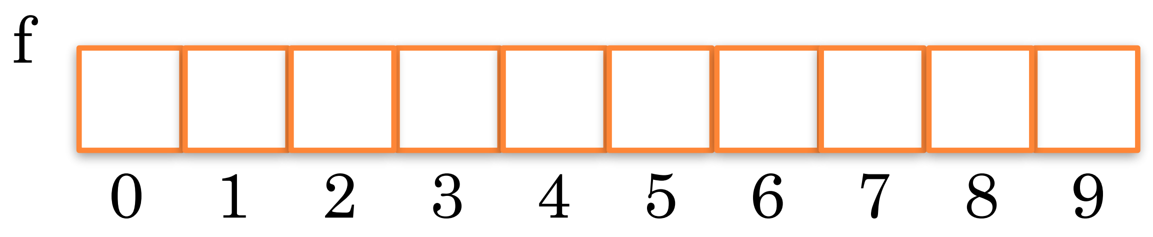
Arrays are collections of variables which have

The same type and share some of the same name.

An example

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

This declares an array called f who can hold integers and which has 10 places. The place names begin at 0 and go up to 9.



We refer to the 1st location as f[0] and the last location as f[9]

Review: arrays

Suppose we have declared

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

Suppose we have already read values into f.

Now we are asked to construct a program to sum the values in the array f. We would like to finish with the sum of all the values in f contained in the integer variable called sum.

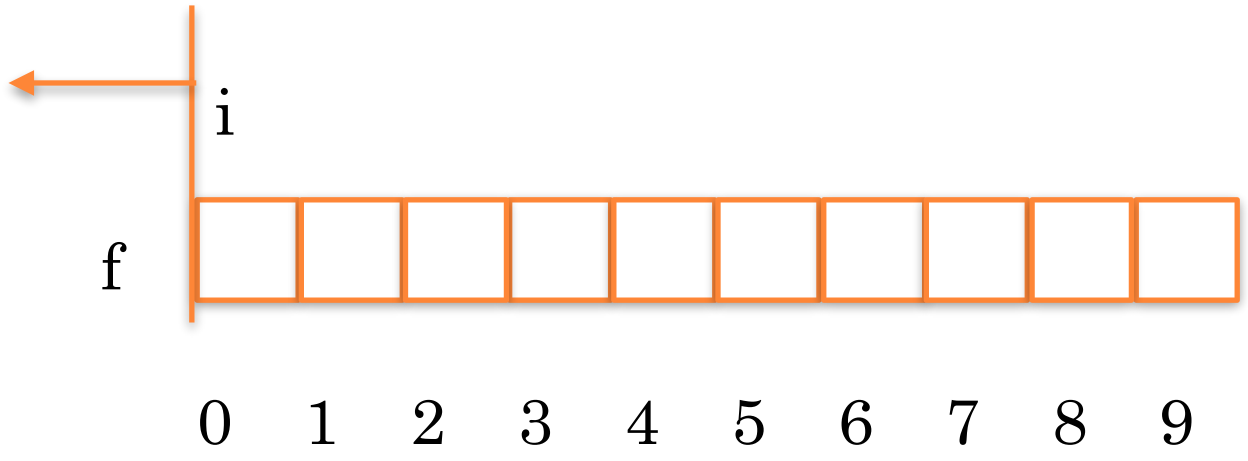
How would we proceed?

Well, as before, let us analyse the problem.

Review: arrays

At **the start**

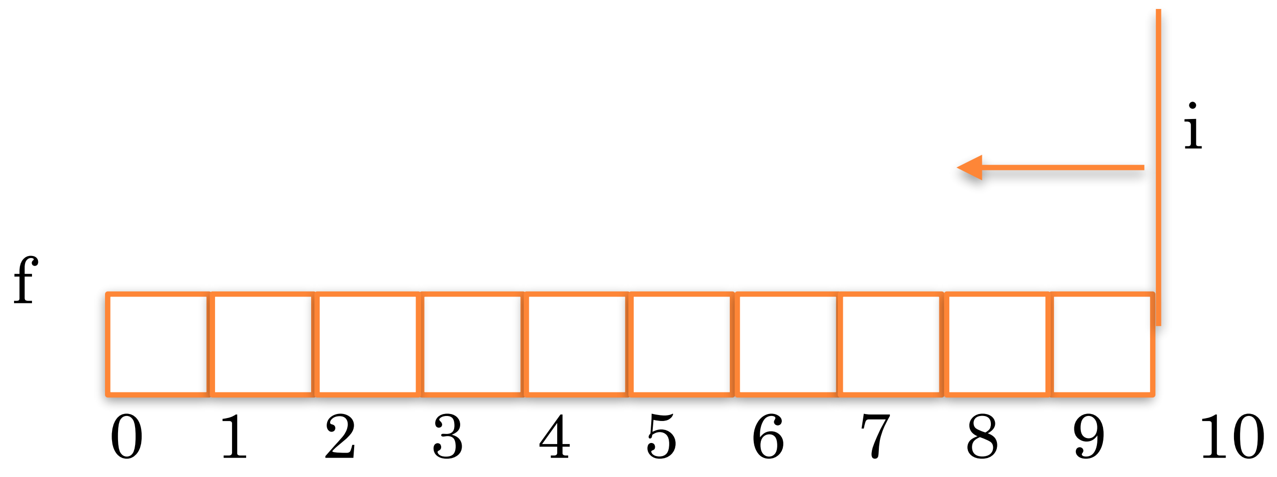
All the values in f to the left of this line have been added and stored in the variable sum



Review: arrays

At **the end**

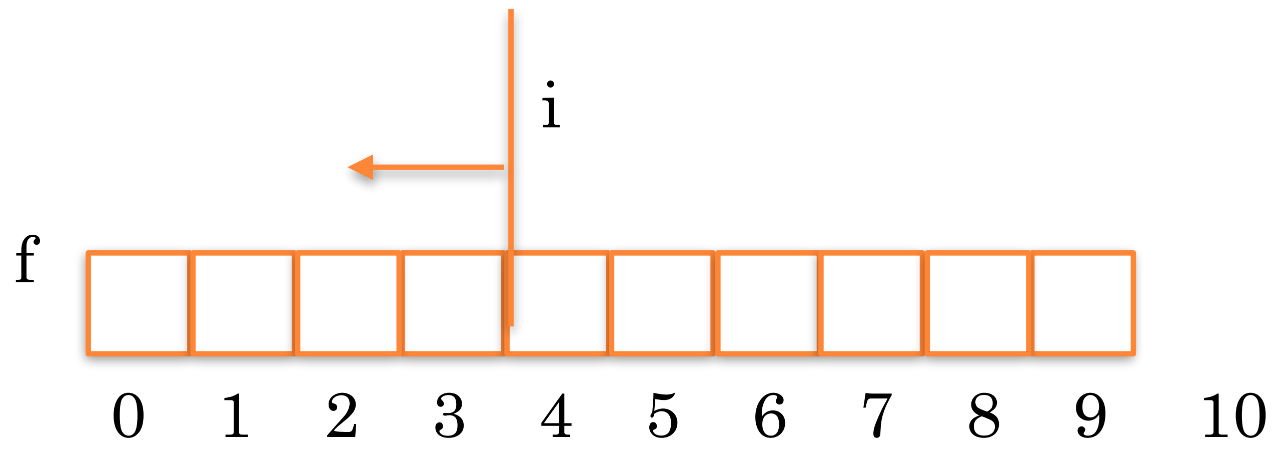
All the values in f to the left of this line have been added and stored in the variable sum



Review: arrays

Somewhere **in the middle**

All the values in f to the left of this line have been added and stored in the variable sum

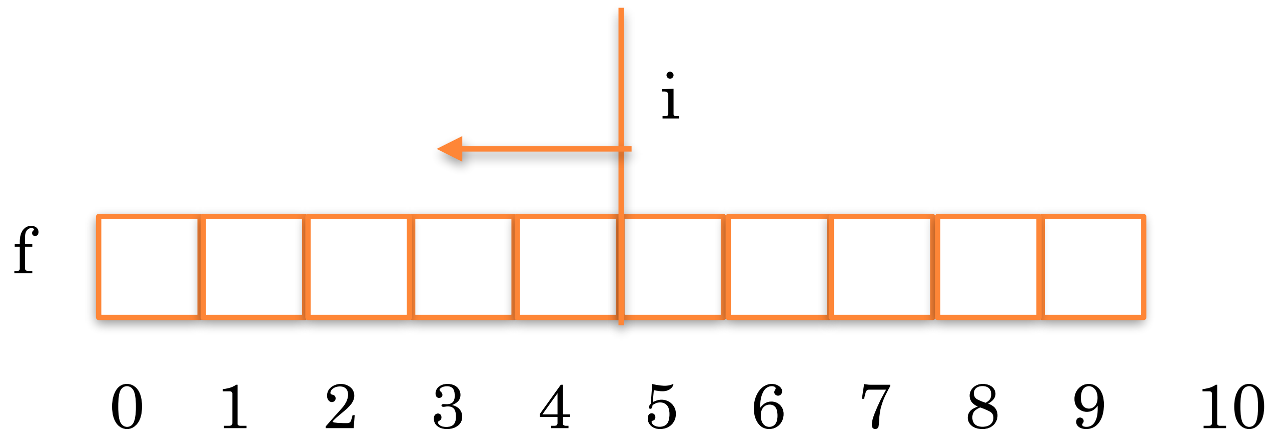


What should we do next?
Add the value in f[4] to sum and move the line one place to the right

Review: arrays

Somewhere **in the middle**

All the values in f to the left of this line have been added and stored in the variable sum



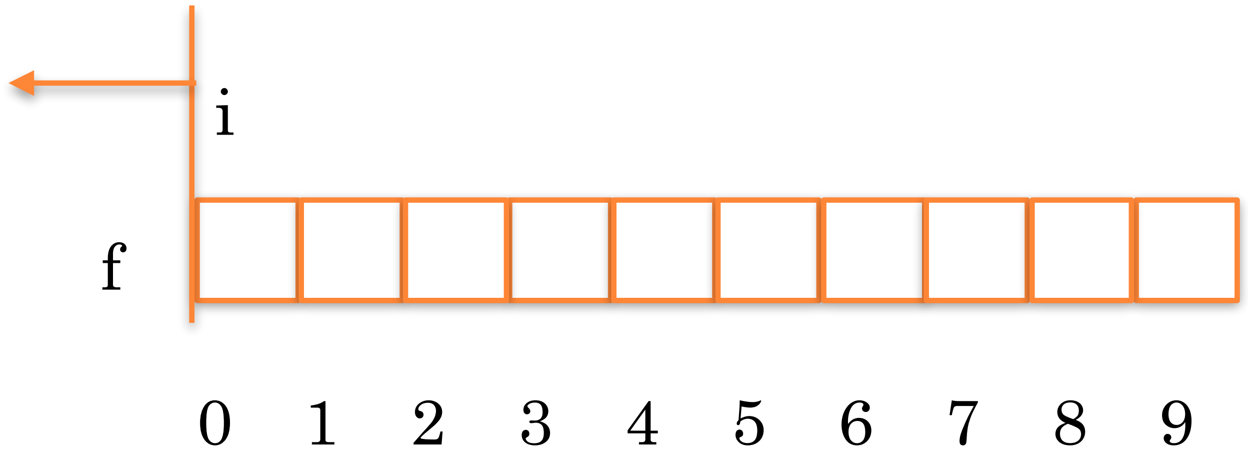
We have added the value in f[4] to sum and moved the line one place to the right

Review: arrays

At **the start**

All the values in f to the left of this line have been added and stored in the variable sum

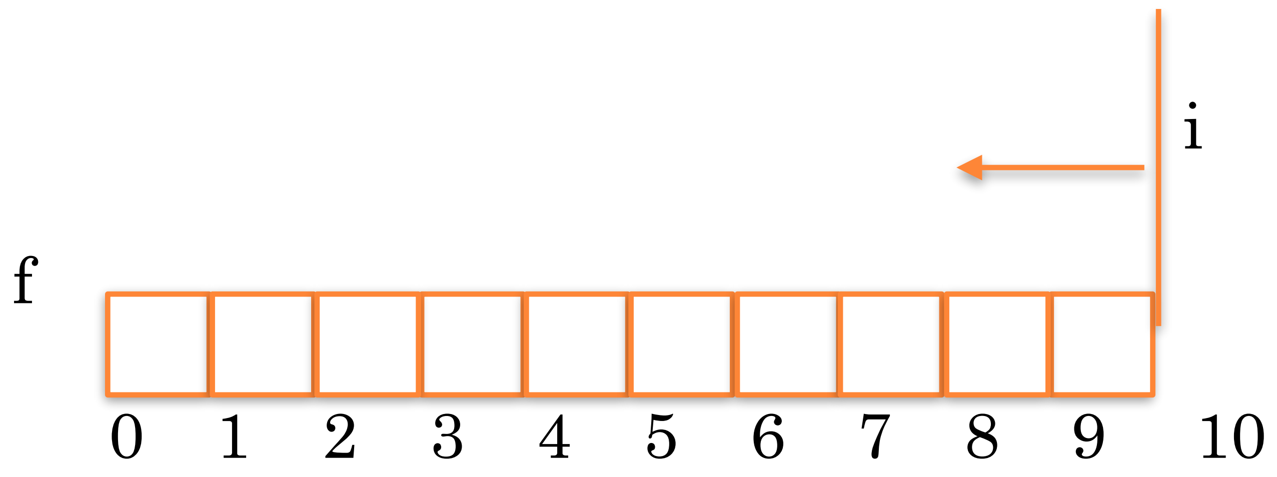
```
sum = 0;
i = 0;
```



Review: arrays

At **the end**

All the values in f to the left of this line have been added and stored in the variable sum

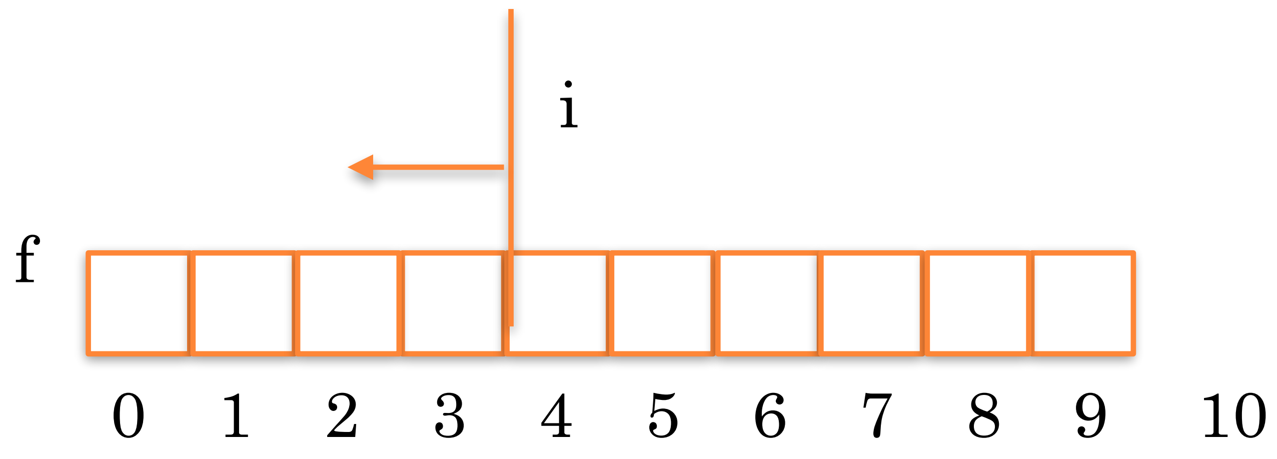


At this point we are finished so i == 10
So when we are not finished the guard on the loop will be i != 10

Review: arrays

Somewhere **in the middle**

All the values in f to the left of this line have been added and stored in the variable sum



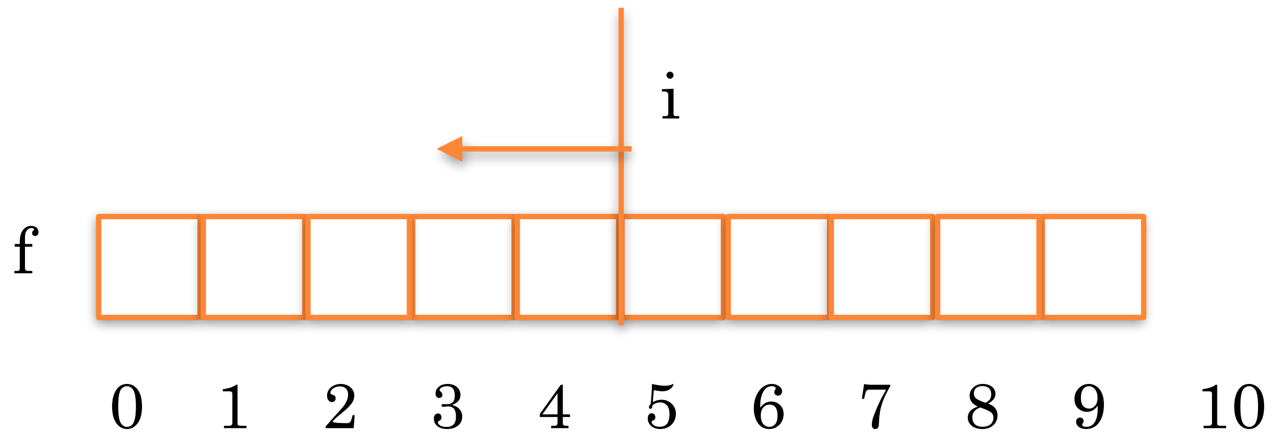
What should we do next?
Add the value in f[4] to sum and move the line one place to the right

Review: arrays

Somewhere **in the middle**

All the values in f to the left of this line have been added and stored in the variable sum

sum = sum + f[i] ;
i = i + 1 ;



We have added the value in f[4] to sum and moved the line one place to the right

```
/*Finished program */
#include <stdio.h>
main ()
    {
    int f [10] ;
    int i ;
    int sum ;

/* code to read values into f */
    sum = 0 ;
    i = 0 ;
    while ( i != 10 )
        {
        sum = sum + f[i] ;
        i = i + 1 ;
        }
    printf ( “the sum of the values in f: %d% \n”,
    sum );
    }
```

Problems

- Given an int array which contains 20 elements (int f[20]),
- Construct programs to do the following
- Get the product of the values
- Get the sum of the values in the 2nd half of f
- Multiply the middle 10 elements together
-

Common shapes

```
i = “start place” ;
result = "some base value" ;
while ( i != “beyond the end”)
        {
        result =  result “combined with f[i]” ;
        i = i + 1 ;
        }
```

This is a very common shape and we call it a **reduction**.

Problems

- Given int f[20] which already contains values, write programs to do the following
- Count the number of even values in f
- Multiply all the negative values in f
- Find out what is the largest value in f

Count the even values in f.

```
int i ;
int count ;

i = 0 ;
count = 0 ;

while (i != 20)
      {
      if (f[i] % 2 == 0)
            { count = count + 1 ; }
      i = i + 1 ;
      }

// count is the number of even values up to position i
// and i == 20
```

Multiply the negative values in f.

```
int i ;
int product ;

i = 0 ;
product = 1 ;

while (i != 20)
        {
        if (f[i] < 0)
                { product = product * f[i] ; }
        i = i + 1 ;
        }

// product is the product of the values up to position i
// and i == 20
```

Compute the largest value in f.

```
#include <limits.h>

int i ;
int large ;

i = 0 ;
large = INT_MIN;

while (i != 20)
      {
      if ( large < f[i] )
            { large = f[i] ; }
      i = i + 1 ;
      }

// large is largest value up to position i and i == 20
```

Another way we could do this would be
to write a function to return what value to use
in each case.

```
int g (int x)
      { if ( x % 2 == 0)
            { return 1 ;}
        else
            { return 0 ; }
      }
```

Count the even values in f.

```
int i ;
int count ;

i = 0 ;
count = 0 ;

while (i != 20)
      {
      count = count + g (f[i] ) ;
      i = i + 1 ;
      }

// count is the number of even values up to position i
// and i == 20
```

Similarly, with computing the product of the negative values
we would have defined the function g like this

```
int g (int x)
      {
      if ( x < 0 )
            { return x ; }
      else
            { return 1 ; }
      }
```

Multiply the negative values in f.

```
int i ;
int product ;

i = 0 ;
product = 1 ;

while (i != 20)
      {
      product = product * g( f[i] ) ;
      i = i + 1 ;
      }

// product is the product of the values up to position i
// and i == 20
```

Similarly, with finding the largest value in f
we would have defined the function g like this

```
int g (int x, int y)
      {
      if ( x <= y )
            { return y ; }
      else
            { return x ; }
      }
```

Find the largest value in f.

```
int i ;
int large ;

i = 0 ;
large = INT_MIN ;

while (i != 20)
      {
      large = g( large, f[i] ) ;
      i = i + 1 ;
      }

// large is the largest of the values up to position i
// and i == 20
```

These type of problems all have solutions
with a similar “shape”.

In each case we must examine the values in
some range of values.

For each value we must decide whether to use it
or whether to ignore it.

The name for there type of problems is
Partitioned Reduction problems.

You will find them very often. So, learn the shape of the
general solution and always use that shape.

A Challenge

- Construct a program to print out the cubes of the first N natural numbers.
- We will not allow you to use multiplication or exponentiation.