Pointers

Computer memory is a vast collection of **bits**, that can be either 0 or 1. These bits are grouped into 8-bit blocks called **bytes**. Bytes are then grouped into **words** depending on the word-size of the computer: a 32-bit system has four bytes per word, and a 64-bit system has 8 bytes per word.

Every byte in a computer’s memory is uniquely identified by its location. Addresses range from 0 up to the size of the memory, so 2 GB of memory would have addresses that go up to 232 – 1.

We can find out the address of a variable (the address of the first byte in the memory that the variable is using) by using the address operator, &. (You may remember this operator from the scanf() function.) In the example below we use the address operator to print the address of the variable foo, and %p to format it as a memory address.

int foo = 10;

printf("The address of foo is %p and its content is %d.", &foo, foo);

Output (example address):

The address of foo is 0x22cca4 and its content is 10.

A variable that stores the address of another variable is called a **pointer**. There are different pointer types for each variable type: a pointer to the address of an int variable is not the same variable type as a pointer to the address of a float variable.

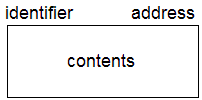
We declare pointer variables using the asterisk operator and the type of variable it is pointing at.

int i; /\* i can only store integer values \*/

int \*p; /\* p can only store the address of an integer \*/

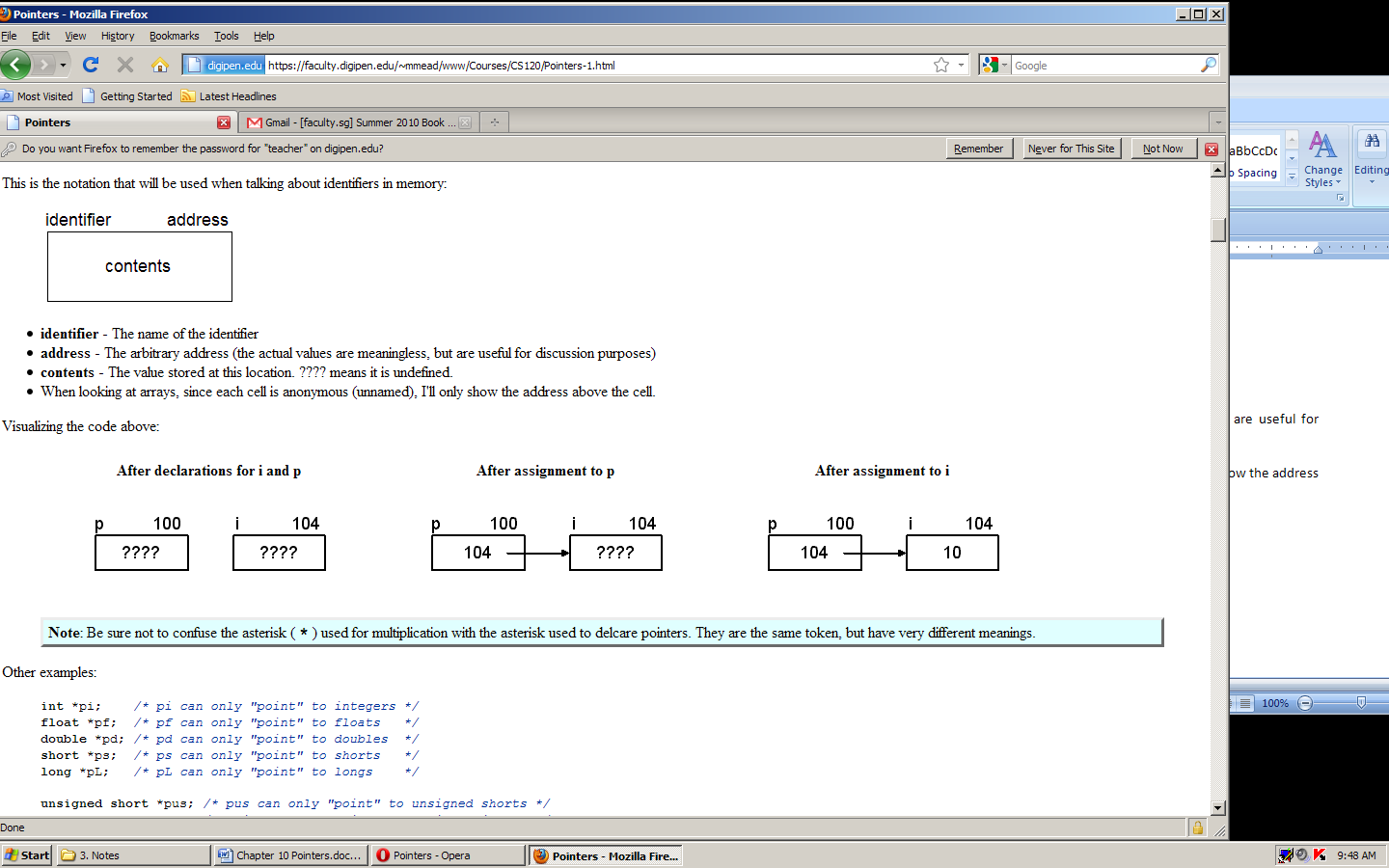
p = &i; /\* the value of p is now the address of i \*/

i = 10; /\* the value of i is now 10 \*/

This is the notation we will use when talking about variables in memory:

* **identifier** – the name of the variable
* **address** – the address of the memory (we will be using example values)
* **contents** – the value stored at this memory location

Visualizing the code above with this type of notation looks like this:



If we need to store a pointer that is pointing at nothing, we can set it to NULL, like this:

int \*p = NULL;

This is much safer than leaving the pointer uninitialized. A pointer set to NULL is like setting an integer to zero, it is an actual value that can be checked (such as p != NULL), and it will never be confused with an actual memory address. An uninitialized pointer can have a random value that looks like a memory address, and perhaps is an actual memory address, which can lead to reading or writing to a random location. This can cause some very annoying bugs in your program.

The asterisk has yet another meaning as the **dereference operator**. This is how we access the value of whatever memory the pointer is pointing at.

|  |  |
| --- | --- |
| **Code** | **Output** |
| int i = 10;  int \*p = &i;  printf("Value of i is %d\n", i);  printf("Address of i is %p\n", &i);  printf("Value of p is %p\n", p);  printf("Address of p is %p\n", &p);  printf("Value of what p is pointing  at is %d", \*p); | Value of i is 10  Address of i is 0x22cca4  Value of p is 0x22cca4  Address of p is 0x22cca0  Value of what p is pointing at is 10 |

Once we have a pointer to a variable, we can both read and write the data through that pointer.

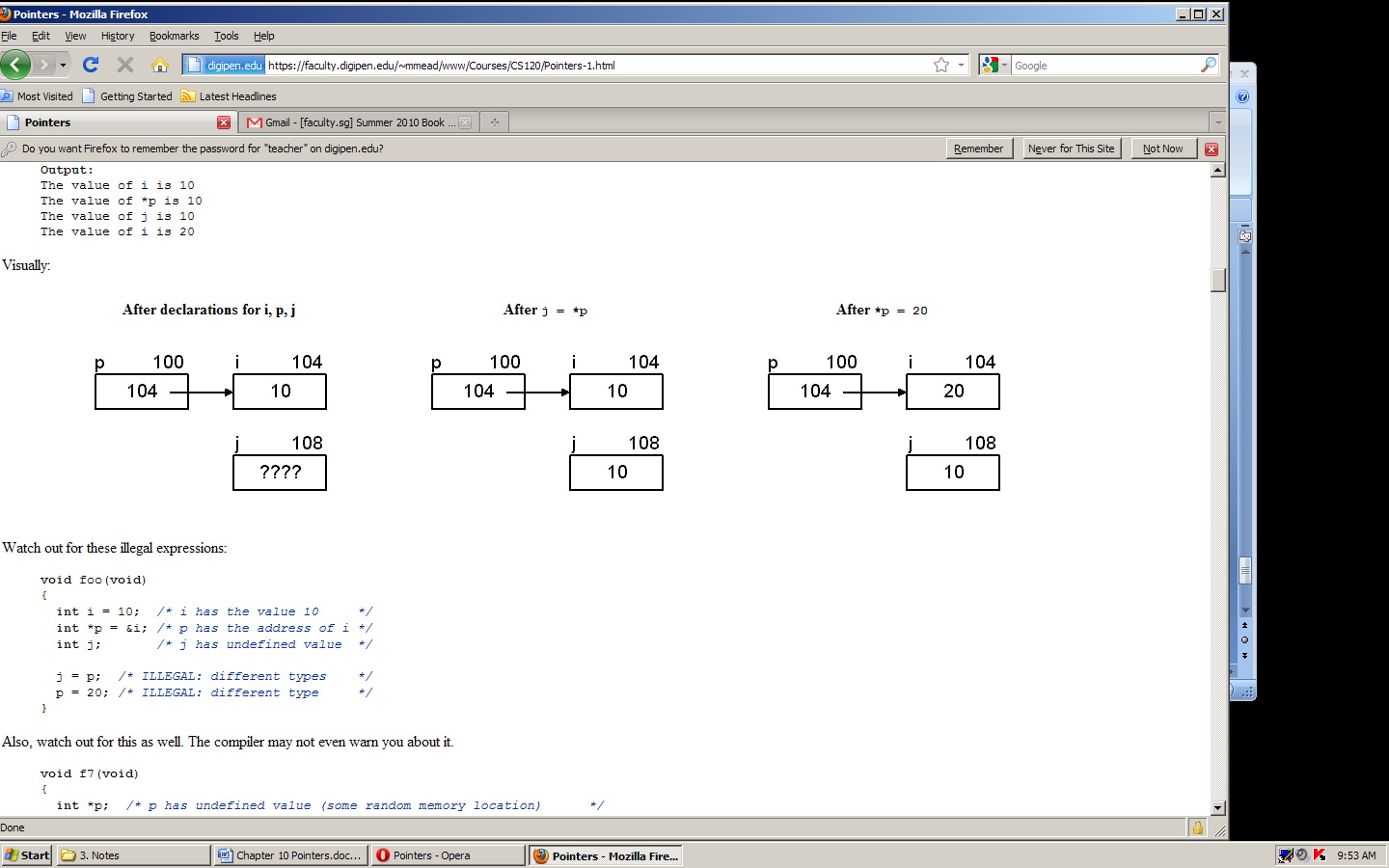
int i = 10; /\* i has the value 10 \*/

int \*p = &i; /\* p has the address of i \*/

int j; /\* the value of j is undefined \*/

j = \*p; /\* j has the value of i \*/

\*p = 20; /\* i has the value 20 \*/



# Pointers and Functions

One major benefit of pointers is when we need a function to modify a value that is out of the function’s scope. A simple example we’ve already seen is when using scanf():

int a;

scanf("%d", &a);

We want scanf() to modify our local variable, but since it can’t access it directly we need to pass the address so that it can access the memory.

Remember that function parameters make copies of the data that is passed in. In the example below, calling the bad\_swap function doesn’t affect the original variables, because the function is changing copies of the ints.

void bad\_swap(int a, int b)

{

int temp = a;

a = b;

b = temp;

}

int main(void)

{

int x = 10;

int y = 20;

printf("Before: x = %i, y = %i\n", x, y);

bad\_swap(x, y);

printf("After: x = %i, y = %i\n", x, y);

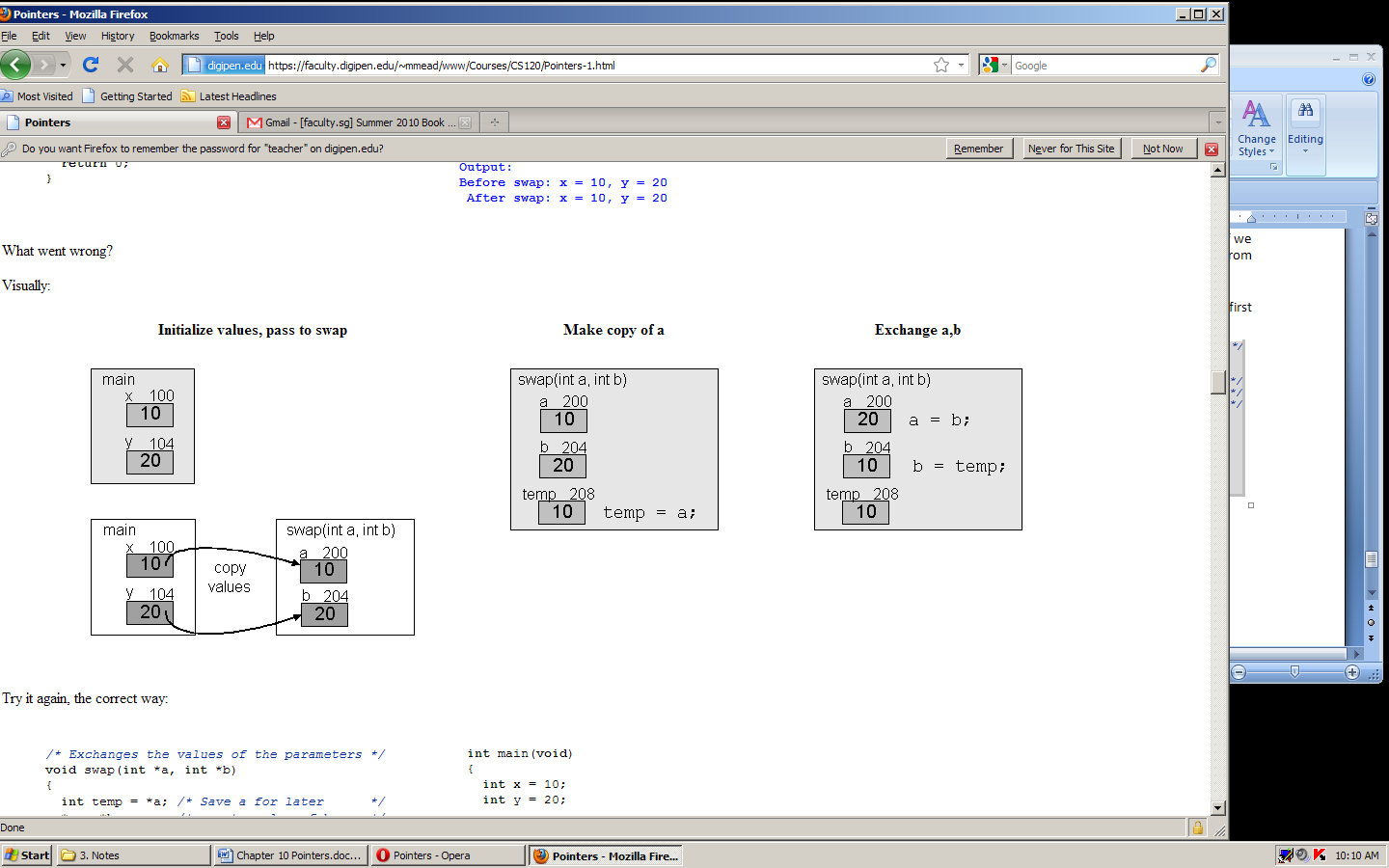
return 0;

}

Output:

Before: x = 10, y = 20

After: x = 10, y = 20



If we re-write that swap function to take pointers to ints instead, we can actually swap the values in the variables.

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main(void)

{

int x = 10;

int y = 20;

printf("Before: x = %i, y = %i\n", x, y);

swap(&x, &y);

printf("After: x = %i, y = %i\n", x, y);

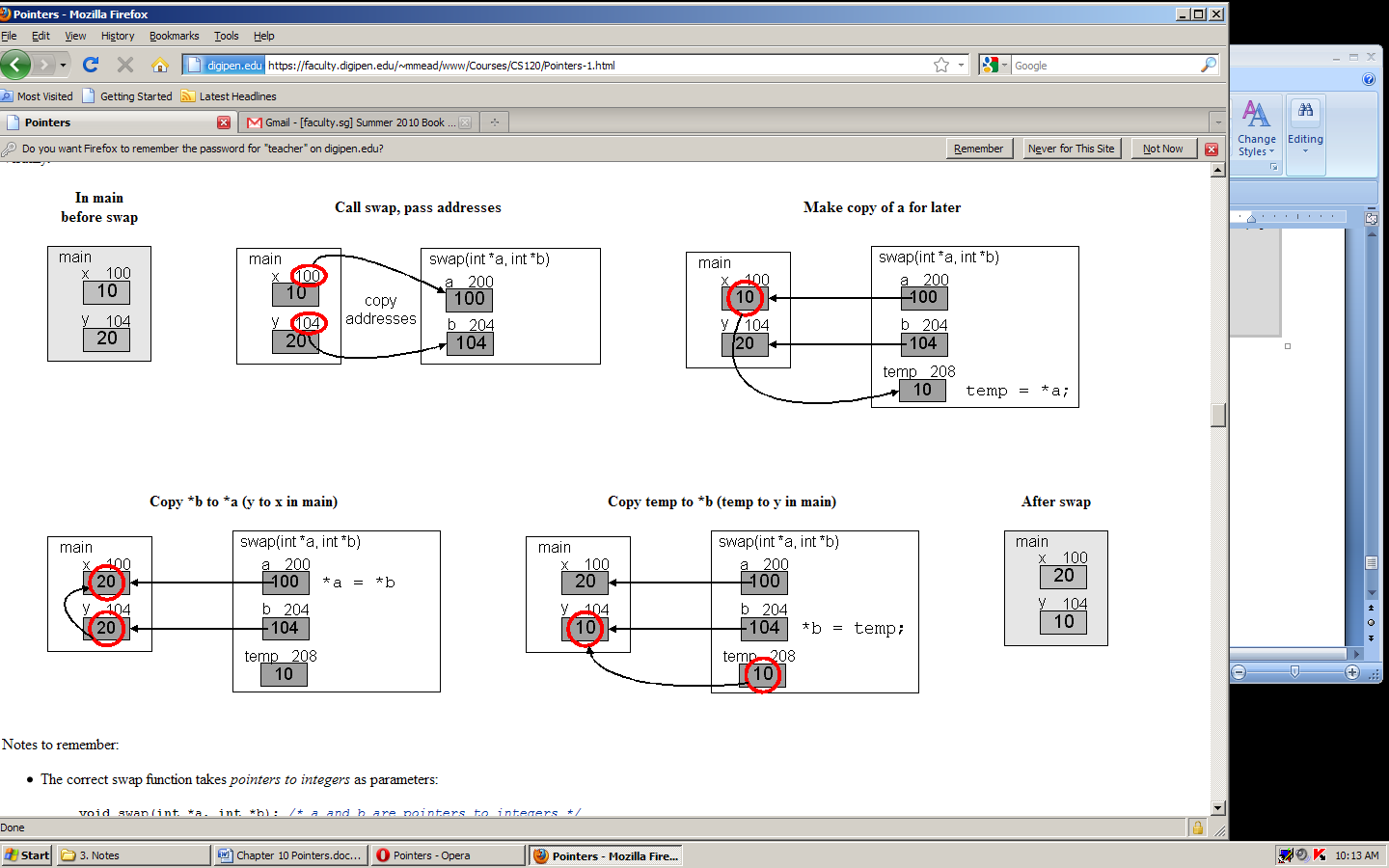
return 0;

}

Output:

Before: x = 10, y = 20

After: x = 20, y = 10



# Pointer Operations with Arrays

We can do some limited arithmetic operations with pointers, but these operations only make sense when we are dealing with arrays.

* pointer + integer = pointer
* pointer – integer = pointer
* pointer – pointer = signed integer

When we add a number to a pointer, the compiler is essentially moving the pointer that many steps forward in memory, each step being the size of the type of data that it is pointing at. If our pointer starts out by pointing at the first element in an array, adding one to the pointer moves it to point at the second element in the array. We can also use the ++ and -- operators to quickly add or subtract one, just like with integers.

In the example below, we create an array of integers and a pointer to integers, set the pointer to point at the first element in the array, and use the pointer to set the value. The loop will then continue to set the value of the rest of the elements.

int a[5];

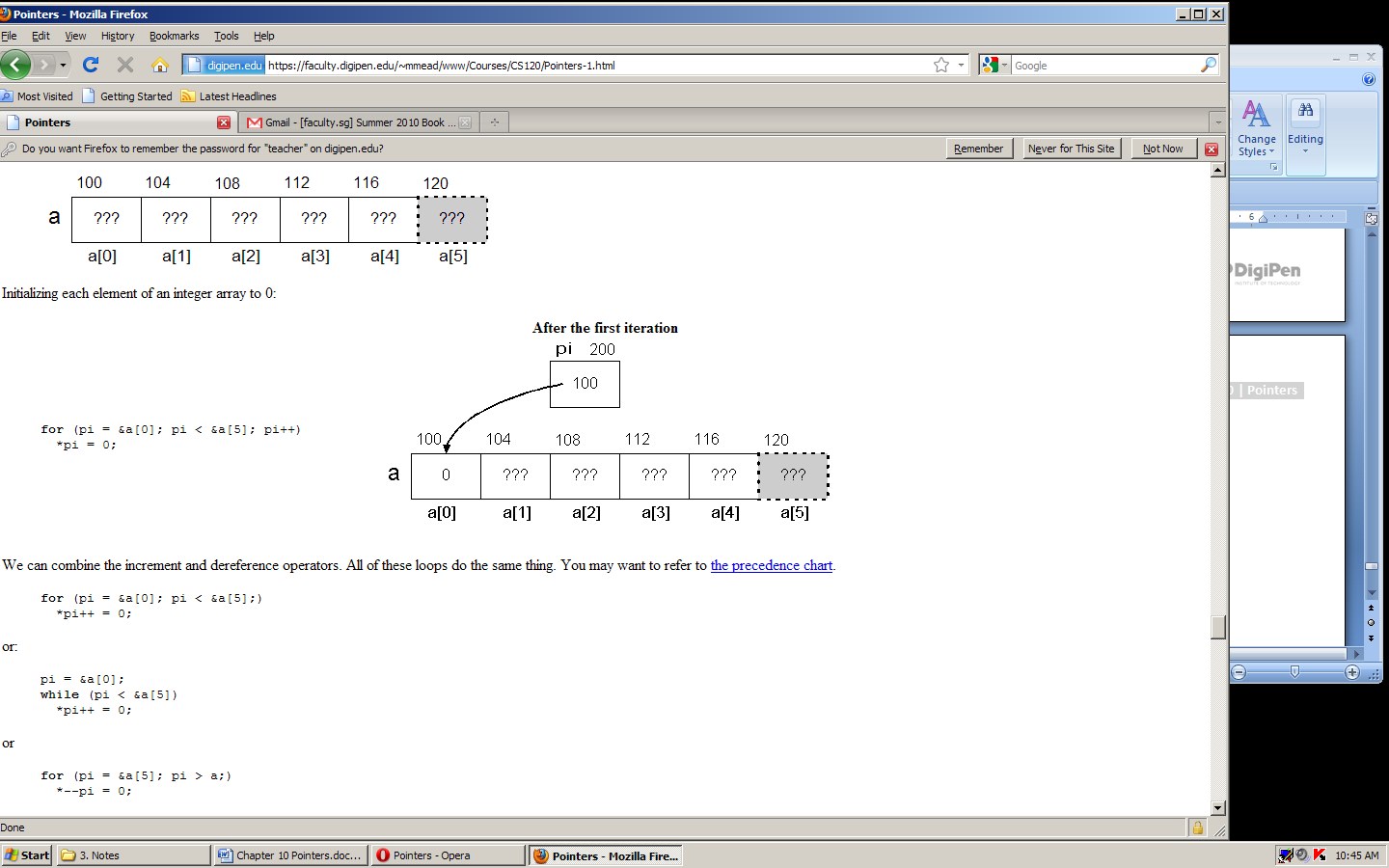
int \*pi;

for (pi = &a[0]; pi < &a[5]; pi++)

{

\*pi = 0;

}



Notice that in the for loop we are comparing pi against the address of a[5], which is past the end of the array. This is okay because we are simply checking the memory address. What we can’t do is dereference the pointer once it’s pointing at that memory.

We could also write the same loop in a different way, using pointer arithmetic to access each array element.

for (pi = &a[0], int i = 0; i < 5; i++)

{

\*(p + i) = 0;

}

Subtracting two pointers that are pointing at the same array will tell us the difference in elements between the two elements that the pointers are pointing at.

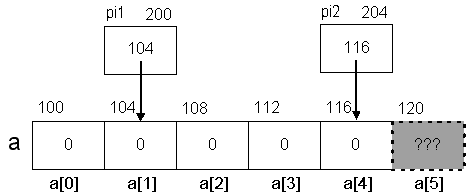
int a[5];

int \*pi1 = &a[1];

int \*pi2 = &a[4];

int diff = pi2 – pi1; /\* diff is 3 \*/

diff = pi1 – pi2; /\* diff is -3 \*/



# Comparing Pointers

If two pointers are pointing at the same array, we can use the comparison operators <, <=, >, and >= to check the relative positions of the elements the pointers are pointing at.

We can use the equality operators, == and !=, with any arbitrary pointers. This simply checks whether the memory address stored in the pointers is the same or not.

# Pointers vs. Arrays

As we did above, if we want to access an array with a pointer we can do this:

int \*pi = &a[0];

Now, if we want to access the third element in the array, we can do it two ways:

int value = \*(pi + 2); /\* value is 3 \*/

value = a[2]; /\* value is still 3 \*/

Even though a is not a pointer, it can be treated in many of the same ways. To set our pointer to the beginning of the array we can also do this:

int \*pi = a;

And, to access an array element, we can use a in much the same way as we use pi.

int value = \*(a + 2); /\* value is 3 \*/

In general, array[index] is the same as \*(array + index), where array is an array of any type and index is any integer expression.

This means that the following subscript and pointer expressions are equivalent.

a[i] ==> \*(a + i)

&a[i] ==> &(\*(a + i))

&a[i] ==> &\*(a + i)

&a[i] ==> a + i

a[i] ==> \*(a + i)

a[0] ==> \*(a + 0)

a[0] ==> \*a

&a[0] ==> &\*a

&a[0] ==> a

When passing arrays into a function, void function(int array[]) is exactly the same as void function(int \*array). Inside that function, you could access array using either subscripts or pointer arithmetic.