Using Pointers

Section 4.2

Introduction

- Several common uses of pointers
  - Output parameters in functions
  - Working with arrays
  - Dynamic memory allocation
  - Pointers to pointers (double pointers)

Pass by Value

- By default, C uses **pass by value** (like Java)
- When a variable is passed into a function, the value in the variable is copied into the corresponding function parameter
  - i.e., a copy is made of the variable
- Since the parameter contains a copy, modifying the parameter does not alter the variable passed into the function

```
void f(int y)
{
    y = 5;
}

int main(int argc, char *argv[])
{
    int x = 100;
    f(x);
    printf("%d", x);
    return 0;
}
```

Pass by Reference

- Another way of passing is **pass by reference**
- Instead of passing a value that's in variable into a function, the address of the variable (reference) is passed into the function
- Effect is that changes made to the variable inside the function also change the variable originally passed in

- To make a function parameter **pass by reference**, make its data type a **pointer** when creating the function
  - **void f(int *y)**
- Now, a memory location is being passed into the function

- To pass a value into a function by reference, often use the **reference operator** (**&**) on a variable
  - **f(&x);**
  - Doing so passes the address of the variable into the function's parameter

```
void f(int *y)
{
    *y = 5;
}

int main(int argc, char *argv[])
{
    int x = 100;
    f(&x);
    printf("%d", x);
    return 0;
}
```

Output Parameters

- **return** keyword allows us to specify a value to return from a function
  - Can only specify a single value each call
- Generally, use function parameters to hold inputs
- However, can use **pass by reference** to have function parameters to hold outputs as well
  - Calling the function has the effect of filling the output parameter

Example

- **division.c**
  - Uses pass by reference to output two values from a single function call
  - Notices the same and different memory locations

Label	Address	Value
numerator	400 – 403	9
denominator	404 – 407	2
dividend	408 – 411	708
remainder	412 – 415	712
x	700 – 703	9
y	704 – 707	2
d	708 – 711	4
r	712 – 715	1

Pointers and Arrays

- In C, an **array variable** is essentially a **pointer**
  - Both hold a memory location
- **Array indexing** works the same as **pointer arithmetic**

- Suppose we have an array called array
  - **int array[10];**
- Can be indexed into several ways
  - **array[5]** is really equivalent to ***(array + 5)**
  - Thus, **5[array]** will also work
    - ***(5 + array)**

- Important difference is that a pointer's value (memory address) can change
  - Pointer can be set to point to a different memory location
- An array variable's memory address cannot change

Example

- **arrays_pointers.c**
  - Gets index and value from user
  - Puts value at that index
  - Demonstrates various ways of indexing

Dynamic Memory Allocation

- When a program is compiled, sees variables and knows how much memory is needed for them
  - Known as **static memory allocation**
- Sometimes, need to allocate additional memory for variables while program is running
  - E.g., create an array based on a size provided by user
  - Known as **dynamic memory allocation**

- Program has two primary areas of memory: the **stack** and the **heap**
- Statically allocated variables go on the stack
- Dynamically allocated variables go on the heap

malloc()

- **malloc()** is primary function for dynamic memory allocation
  - Input is an integer for number of bytes to allocate
  - Output is a void pointer to that chunk of memory

```
int * ptr;

ptr = (int *)malloc(sizeof(int) * 10);
```

- Asks operating system to allocate specified amount of memory
  - If fails (i.e. not enough memory), returns **null pointer**
    - **null pointer** is a pointer to 0 (nothing)
  - Otherwise, returns a **void pointer** to the chunk of memory
    - **Void pointer** is like a wildcard or generic pointer
    - Cast to type we want

```
double *a;
a = (double *)malloc(40);

if (a == NULL)
    printf("Allocation failed\n");
else
    printf("a at %p\n", a);
```

- Good practice to check that allocation succeeded
  - Trying to use a NULL pointer causes a crash
    - Easy to detect and avoid
- Commonly use **sizeof()** operator with malloc()
  - **malloc(sizeof(int) * 10)**
    - Allocates memory for ten ints

Memory Leaks

- Statically allocated memory is automatically released
  - I.e., when variables scope / block of code ends
- Dynamically allocated memory is not automatically freed
  - Programmer must explicitly release the memory
- Failure to explicitly release the memory leads to a bug called a **memory leak**

```
double *a;
a = (double *)malloc(70);
a = (double *)malloc(300);
```

free()

- **free()** is used to release dynamically allocated memory
  - Input is pointer to the dynamically allocated memory
    - Pointer received from malloc()
    - **free(a);**

Example

- **dynamic_array.c**
  - Dynamically allocates an array from number from user
  - Checks for allocation failure
  - Frees memory at end

Pointers to Pointers

- Pointers to point to the memory location of a variable
- The variable pointed to can actually be a pointer itself
- This is called a pointer to a pointer or a double pointer
  - **int **ptr;**
    - A pointer to a pointer to an integer
    - Still holds a memory location

- A pointer and an array variable have a lot in common
- A **double pointer** has a lot in common with a **2D array**
  - Hence why **char *argv[]** can also be written as **char **argv**
- Basically an array of pointers
  - Each pointer is the start of an array itself

```
double **m;

m = (double **)malloc(sizeof(double *) * 2);

m[0] = (double *)malloc(sizeof(double) * 3);
m[1] = (double *)malloc(sizeof(double) * 2);

m[0][1] = 6.3;
m[1][0] = -2.8;
```

Label	Address	Value
m	400 – 403	10000
m[0]	10000 – 10003	10008
m[1]	10004 – 10007	10032
m[0][0]	10008 – 10015	
m[0][1]	10016 – 10023	6.3
m[0][2]	10024 – 10031	
m[1][0]	10032 – 10039	-2.8
m[1][1]	10040 – 10047	

- Commonly used to change the value of a reference passed into a function
  - For example, give a function a pointer to a pointer, and it makes an array for you

Example

- **double_pointer.c**
  - Function is given address of a pointer
  - Function allocates memory for array

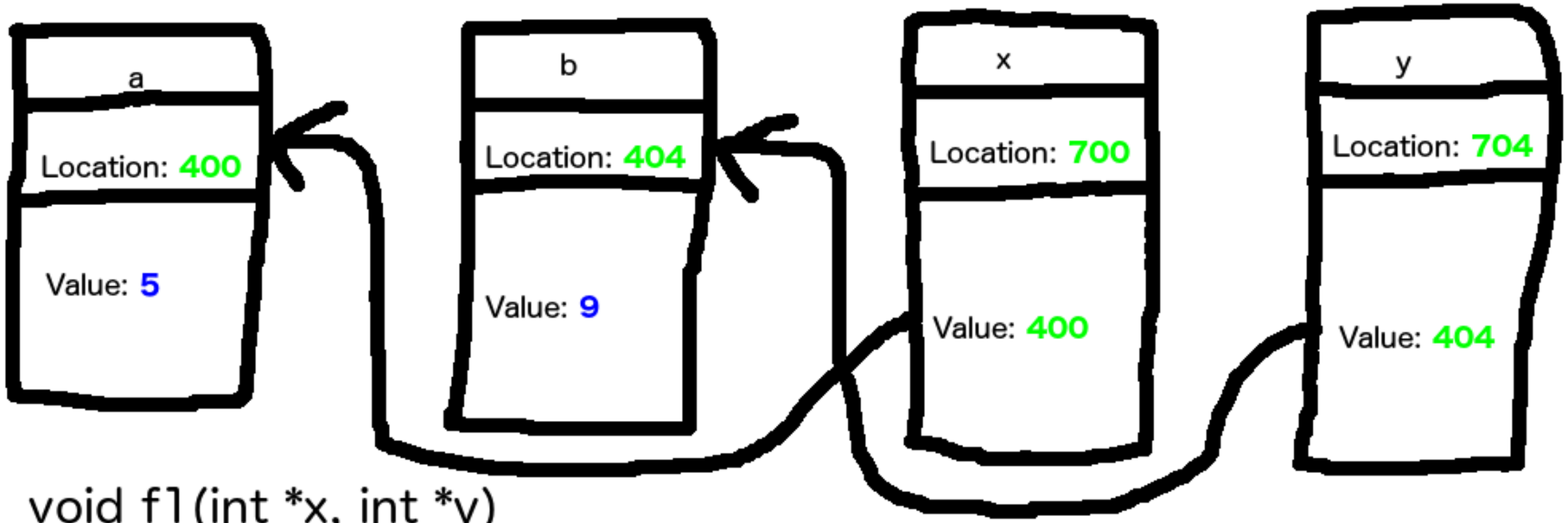
Using Pointers for Output

- Common use of pointers is to output multiple values from a single function call
- Empty variables are passed by reference into the function
- Function's job is to fill up these empty variables

Summary

- Pointers are useful for
  - Outputting values from a function
  - Dynamic memory allocation
- Pointers have a connection with arrays

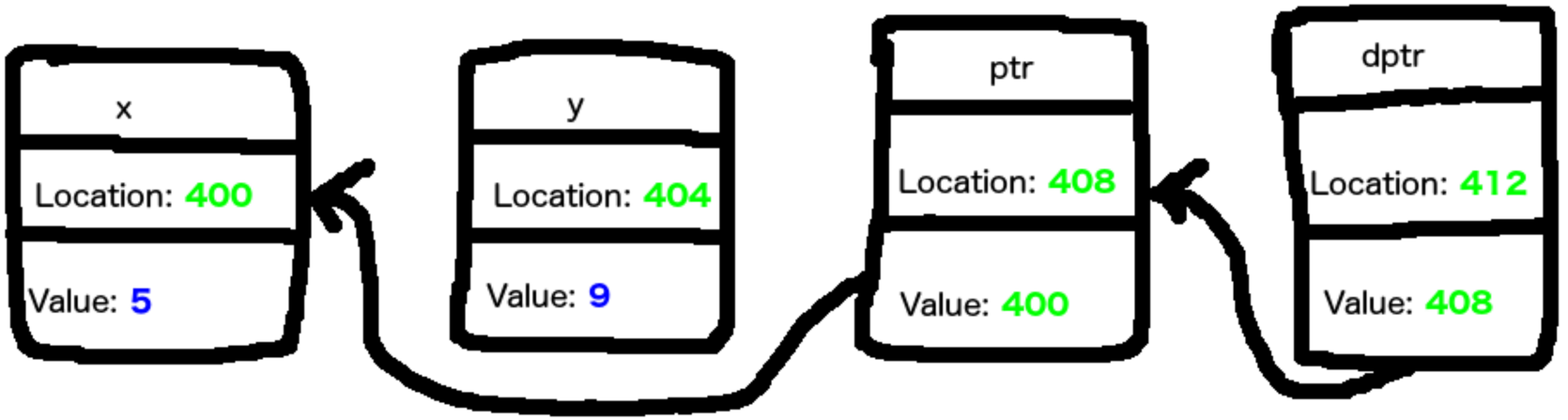
- A double pointer is a pointer to a pointer
- Double pointers are useful for
  - Passing a pointer by reference
  - Working with a 2D array




```
void f1(int *x, int *y)
{
    *x = 5;
    *y = 9;
}

int main(void)
{
    int a;
    int b;

    f1(&a, &b);

}
```

Double Pointers

- Pointers point to the memory location of a variable
- Variables being pointed to can be pointers themselves
- These are known as **double pointers** or **pointers to pointers**

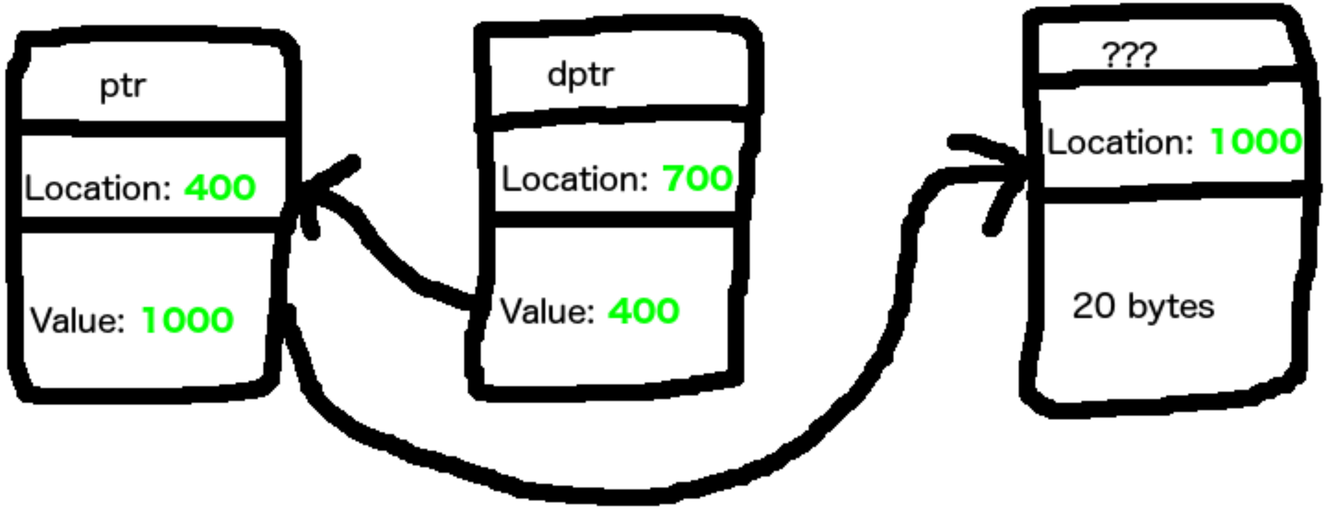



```
int x = 5;
int y = 9;

int *ptr;
int **dptr;

ptr = &x;

dptr = &ptr;
```

```
printf("%d\n", *ptr);
printf("%d\n", &ptr);

printf("%d\n", *dptr);
printf("%d\n", **dptr);
printf("%d\n", &dptr);
```

Passing a Pointer by Reference

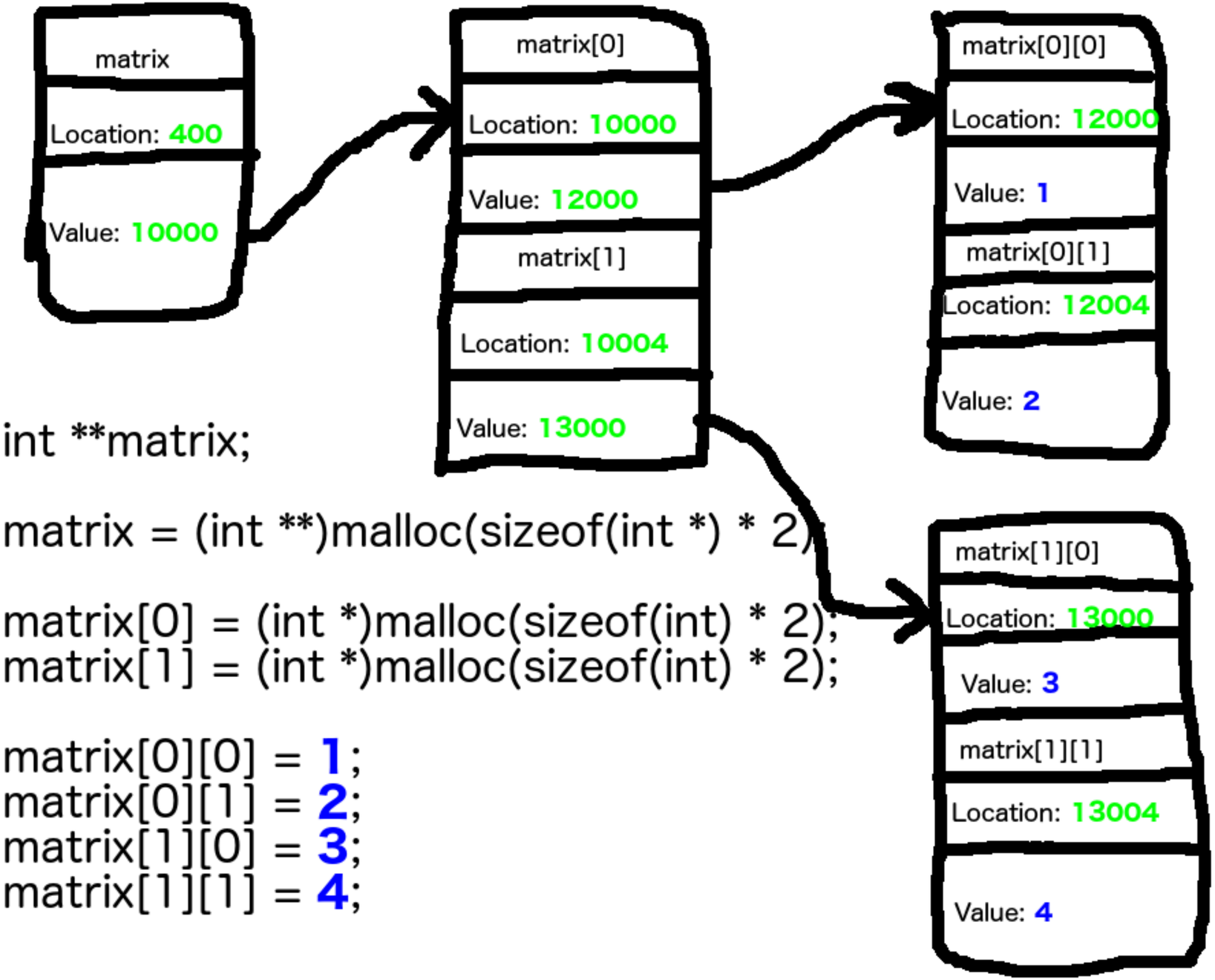
- Common application of double pointers is to have a function fill a pointer
  - Wouldn't work with just a single pointer, would be like pass by value
- Passing a pointer by reference into a function puts it in a double pointer in the function




```
f(int **dptr)
{
    *dptr = (int *)malloc(sizeof(int) * 5);
}

main()
{
    int *ptr;

    f(&ptr);
}
```

Double Pointer for 2D Array

- Another application of double pointers is dynamically allocated 2D arrays
- Double pointer like an array of pointers
  - Each element is a pointer to a 1D array




```
int **matrix;

matrix = (int **)malloc(sizeof(int *) * 2);

matrix[0] = (int *)malloc(sizeof(int) * 2);
matrix[1] = (int *)malloc(sizeof(int) * 2);

matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[1][0] = 3;
matrix[1][1] = 4;
```