Week 6 Notes – Pointers

Cody Fogel, Colt Currie, Div Patel, and Ellie Swaim

Table of Contents:

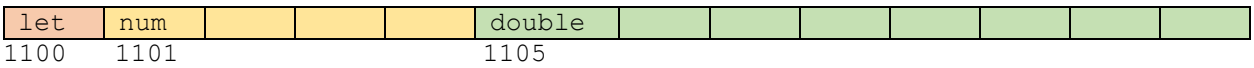
Chapter 10: Pointers

1. Pointers

1.A Pointers and the Address Operator

All variables are allocated enough memory to hold values of their type. Each variable's memory address is the memory address of the first byte allocated to it. The address operator & can be added before a variable to retrieve the memory address of that variable. Memory addresses are always in hexadecimal. A value that holds a memory address is called a *pointer.*

```
char let;       // chars are allocated 1 byte of memory
int num;        // shorts are allocated 4 bytes of memory
double decimal; // doubles are allocated 8 bytes of memory
```



1.B Pointer Variables

A *pointer variable* is a variable that holds a memory address, but it's usually just called a *pointer.* When defined, it must include the type of data that it will point to.

```
double* newPtr; //this pointer will point to the address of a double
```

The * is called the *indirection operator* because it allows you to shift from a pointer to the variable being pointed at. You can use pointers to indirectly access variables by *dereferencing* them. If you dereference a pointer, you can work with the value the pointer points to.

```
double* newPtr;  // this pointer will point to the address of a double
double x = 15.6;

newPtr = &x;     // newPtr will point to the memory address of x

*newPtr = 1.9;   // assigns 1.9 to memory address pointed to by newPtr (aka x)
```

2. Arrays and Pointers

2.A The Relationship Between Arrays and Pointers

An array name represents the memory address of the first element of the array, so it is a pointer. You can use the indirection operator to dereference it and access the first value of the array. Array names are *pointer constants,* however, and cannot be changed after they are created.

```
int arr[] = { 1, 2, 3, 4 };

cout << arr;  //prints the memory address of the first array element
cout << *arr; //prints 1
```

Array elements can be accessed using the indirection operator and pointer addition. The rule is that `array[index]` is equivalent to `*(array + index)`.

```
cout << *(arr + 2); //prints 3
```

Pointers can also be used as array names.

```
char letters[] = { 't', 'r', 's' };

char* letterPointer = letters;
cout << letterPointer[2]; //prints s
```

2.B Pointer Arithmetic

You can also perform basic math with pointers. This is mostly helpful in scrolling through arrays.

```
char* letterPointer = letters;

letterPointer++;
cout << letterPointer; // prints (letterPointer+1) and onward

letterPointer--;
cout << letterPointer; // prints (letterPointer) and onward
```

You can also use operators such as += or −=. Use of these operators is similar to `*(array + index)`, or, if you want to go backwards in an array, `*(array – index)`.

2.C Initializing Pointers

Pointers can be declared with other values of the same type, or with arrays of the same data type. They can also be assigned the memory address 0. A common way to refer to memory address 0 is with the `NULL` constant. In C++ 11, the keyword `nullptr` can be used. To declare a null pointer. You can also use curly braces after declaring a variable to set it to its default value, which for pointers is `nullptr`.

2.D Comparing Pointers

Pointers can be compared just like variables can, using all of the same operators. Since array elements are stored in separate blocks of memory, you can compare pointers to check values in arrays. Most comparisons involving pointers are comparisons to 0, `NULL`, or `nullptr`.

```
int vals[] = { 10, 20, 30, 40 };
int* valPtr = vals;

//Valid comparisons follow

while (*valPtr < vals[3]) {
	cout << *valPtr;	//prints the value being pointed to by valPtr
	valPtr++;		//changes value valPtr points at to the next value in vals.
}

if (valPtr != nullptr) {
	cout << "This pointer points to a valid memory address." << endl;
}
```

2.E Pointers as Function Parameters

Pointers can be used as function parameters. They give functions access to original arguments, a lot like reference parameters. To work with variables pointed at by pointer parameters, the pointers must be dereferenced. The function call must also include a memory address argument instead of a regular variable argument.

```
void printUserInput(int* value); //function prototype with a pointer as the parameter
using namespace std;

int main() {

	int number = 6;

	printUserInput(&number);  //a call to a function with a pointer as a parameter

	return 0;
}

void printUserInput(int* value) {

	cout << *value << endl;   //how a pointer parameter could be used in a function
}
```

You can also pass arrays to functions via pointers. This works the same way as passing a variable via pointers, and you can access the array as you usually would with a pointer.

3. Memory Allocation and Pointer Objects

3.A Pointers to Constants and Constant Pointers

Pointers to Constants

You can declare pointers to constant values. If you do this, you can't change the value you're pointing at. You will get a compile-time error if you try to pass a `const` value into a pointer that hasn't been declared as a pointer to a constant.

```
const int* constPtr; //this is a pointer to an int constant
```

A pointer to a constant can also receive the values of non-constant variables. This can be useful in functions where you want to access data but not modify values.

Constant Pointers

Constant pointers are pointers that cannot be changed after they are initialized with a memory address. They must be initialized when they are defined, unless they are a function parameter (they will be initialized upon function call in that case). The address of the pointer cannot change, but the data pointed at can change.

```
int num = 0;
int* const intPtr = &num; //how you define and initialize a const pointer

intPtr = NULL;            //invalid, cannot change the address of intPtr

*intPtr = 5;              //valid, you can change the data intPtr points at
```

Constant Pointers to Constants

These are a combination of constant pointers and pointers to constants. The addresses that they contain cannot be changed, nor can the data that they point to be modified.

```
char letter = 'c';
const char* const letterPtr = &letter; // a constant pointer to a constant
```

3.B Dynamic Memory Allocation

Dynamic Memory Allocation is when you request memory from the computer to create variables while running. The only way the program has to access these memory bytes are through their memory address, so pointers are needed. Programs request memory for new variables using the `new` operator.

```
int* intPtr = nullptr; //initialization of the pointer to NULL
intPtr = new int;      //pointer requesting memory for new variable
```

You can then access the new variable by using the dereferenced pointer.

Most dynamically allocated variables are arrays, which can then be accessed like normal arrays through pointers.

Each call to `new` allocates storage from a set-aside bit of unused memory called the *heap.* If too much dynamic memory is requested, the heap will become depleted and the program will throw a `bad_alloc` exception. This will usually terminate the program.

When you are done with dynamically allocated memory, you should call the `delete` operator and then set the pointer to `nullptr`.

If you call the `delete` operator but do not set the pointer to `nullptr`, then that is said to be a *dangling pointer*. A *memory leak* is when you don't free dynamically allocated memory after you are done using it. You should only use `delete` with pointers that were used with `new`, though.

```
delete intPtr;     //freeing the memory from dynamic allocation.
                   //If you skip this step, you have a memory leak.

intPtr = nullptr;  //setting the memory address of the pointer to nullptr.
                   //If you skip this step, you have a dangling pointer.
```

3.C Returning Pointers from Functions

Sometimes you want to dynamically allocate memory for an object in a function, create the object, and then return the memory address of your new object. This would call for you to have

```
int* row(int n) {

	int* rowArray = new int[n];

	for (int i = 0; i < n; i++) {
		rowArray[i] = i;
	}

	return rowArray;
}
```

Programs that use dynamically allocated memory should always free that memory whenever possible. There are two helpful rules.

1. The function that allocates memory should also be the function to free it whenever possible.
2. A class that dynamically allocates memory should do so in the constructor and then free the memory in the destructor.

3.D Pointers to Class Objects and Structures

You can declare a pointer to a class the same way you can declare a pointer to any other data type. A pointer to a `Person` class would be defined as:

```
class Person {
	int age, height, weight;
}

Person* personPtr = nullptr; //pointer to Person
Person person;
personPtr = &person;         //personPtr will point to person
```

You can still access members of a class through pointers. However, because the dot operator has higher priority than the indirection operator, you need to use parentheses to force the indirection operator to run first.

```
*personPtr.age = 10;   //incorrect
(*personPtr).age = 10; //correct
```

There is also the option of using *structure pointer notation*. It is simpler and easier to read than (*pointer).member notation.

```
personPtr->height = 64; //structure pointer notation being used
                        //to set a person's height to 64 inches
```

You can dynamically allocate these objects with pointers, and you can also have pointers to class objects as function parameters.

3.E Selecting Members of Objects

Some classes and structs contain pointers as members. These values can still be accessed, but the indirection operator must be used.

```
class Student {
public:
	string m_name, m_class; //Student name and class
	double* m_gpa;          //A student's GPA
}

Student topLee;

cout << *topLee.m_gpa;  //This will display the value pointed to by the gpa pointer.
cout << *topLee->m_gpa; //This will display the value pointed to by the gpa pointer too.
```

Expression	Description
`topLee->gpa`	accesses the member pointer gpa of topLee
`*topLee.gpa`	accesses the value pointed at by the member pointer gpa of topLee
`(*topLee).gpa`	accesses the member pointer gpa of topLee
`*topLee->gpa`	accesses the value pointed at by the member pointer gpa of topLee
`*(*topLee).gpa`	accesses the value pointed at by the member pointer gpa of topLee

4. Smart Pointers

Smart pointers are objects that work like pointers but have the ability to automatically delete dynamically allocated memory that is no longer being used. Smart pointers are said to own or manage the object that they point to. There are three kinds: *unique pointers, shared pointers,* and *weak pointers.* To use them, you must include the memory header.

```
#include <memory>
```

Unique Pointers

A smart pointer is an object that wraps a normal pointer to an owned object. You can define and initialize unique pointers in one line, or you can define them and initialize them separately. You should not define a regular pointer, called a *raw pointer*, and then pass it to a smart pointer variable.

```
unique_ptr<int> ptr1(new int); //correct way to create a unique_ptr object
```

Smart pointers do not support pointer arithmetic, but you can dereference them to access the values stored by the memory addresses.

```
*ptr1 = 10; //assigns 10 to the int pointed to by ptr1
*ptr1 += 2; //adds 2 to the int pointed at by ptr1
```

You cannot initialize a unique pointer with another unique pointer, and you cannot assign a unique pointer to another unique pointer, because that would result in two unique pointers sharing ownership of the same object, which kills the point of the unique pointer (puns intended). There is, however, a move() function, which allows you to transfer ownership from one unique pointer to another.

```
unique_ptr<int> ptr2(new int);
ptr2 = ptr1;       //not OK, will give you errors
ptr2 = move(ptr1); //now ptr2 owns the int ptr1 used to own
```

When move() was called, any object that ptr2 had owned was deallocated, ptr2 took ownership of the int previously owned by ptr1, and ptr1 became an empty object.

You can have unique pointers as function parameters, but in the function call, you must use the move() command. However, if you use pass by reference, you don't need to use the move() command in the function call. You can also return a unique pointer.

Unique pointers delete as they go out of scope. If you need to delete them while they are in scope, you have two options:

```
ptr2 = nullptr;
ptr2.reset();
```

You can use unique pointers to point to arrays, but you need to put brackets in their definition so that the program knows to delete them properly.

```
unique_ptr<int[]> arrPtr(new int[10]); //correct way to define an array unique pointer
```

There is also a get() function for unique pointers. It returns the raw pointer of the object owned by the smart pointer.

Shared Pointers

Shared pointers are smart pointers used to manage dynamically allocated objects that might have more than one owner. You can dereference them and access the values contained at the memory addresses they point to.

A *control block* keeps track of all the references to an object and its raw pointer. It is what is responsible for deleting the owned object when its reference count drops to zero. You should avoid having more than one control block per object. To prevent this, only use the raw pointer to initialize one shared pointer.

The get() function returns the raw pointer of the object managed by the pointer, or null if there is no object being managed. The reset() function empties the calling shared pointer. The use_count() function returns the number of all the pointers that are referring to the same object.

If you want to use a shared pointer to refer to an array, you should use a vector instead. This is because shared pointers cannot run the proper destructors for arrays but can for vectors.

Study Questions

Review the following code:

```
#include <iostream>

int main() {

	int x = 15;
	int* ptrToX = x;

	std::cout << "The value of x is " << *x;

	return 0;
}
```

Question #1:
Will this code run and compile fine? If so, what will it print out?

A) Yes, it will print out “15”
B) No, it will hit a compile time error since x is not a memory address.
C) Yes, it will print out the memory address of x.
D) No, it will hit a compile time error since you’re returning 0 in main()

Question #2:
What is a pointer?

A) A pointer is a variable that holds a memory address.
B) A pointer is a variable that holds a memory address of another variable.
C) A pointer is just an array
D) A pointer is just a variable that points to a string

Question #3:
What are three types of Smart Pointers?

A) Unique pointers, Deletion pointers, Varied pointers
B) Simple pointers, Weak pointers, Strong pointers
C) Unique pointers, Shared pointers, Weak Pointers
D) Override pointers, Shared pointers, Default pointers

Question #4:
What is the correct output for the following code?

```
int test[5] = { 15, 0, 13, 12, 4 };
int* pointer = test;
std::cout << *pointer + 2 << ", " << (*pointer) << std::endl;
```

A) 17, 15
B) 13, 15
C) 0, 15
D) 4, 12

Question #5

In the following code, will std::cout for both lines print the same information? If not, then why?

```
int test[] = { 1, 4, 7, 8, 13 };
int* testPointer = test;
std::cout << test[2] << std::endl;
std::cout << testPointer[2] << std::endl;
```

A) Yes
B) No, testPointer[2] needs a * in front of it to be able to dereference the variable and output the array information.
C) No, test[2] would output 7 while testPointer[2] would output 4

Review the following code:

```
#include <iostream>

struct Height {
	int m_feet, m_inches;
	Height(int feet, int inches) {
		m_feet = feet;
		m_inches = inches;
	}
	Height() {
		m_feet = 0;
		m_inches = 0;
	}
};

class Person {
public:
	std::string m_name;
	int m_age;
	Height* m_height = new Height();

	Person(std::string name, int age, Height height)
	{
		std::cout << "Building a person!!" << std::endl;
		m_name = name;
		m_age = age;
		m_height->m_feet = height.m_feet;
		m_height->m_inches = height.m_inches;
	}
};

int main() {
	Person* Jacob = new Person("Jacob", 18, Height(5,11));

	std::cout << Jacob->m_name << std::endl;
	std::cout << Jacob->m_age << std::endl;
	std::cout << Jacob->m_height->m_feet << '\'' << Jacob->m_height->m_inches;
	std::cout << std::endl;

	delete Jacob;
}
```

Question #6

There is a problem with the code above, it presents a memory leak. What is the easiest way we can fix that?

A) Add a destructor to class Person that will free the memory for m_height when the object for Person is destroyed.
B) Add a destructor to the struct Height that will delete m_feet and m_inches when the object for Height is destroyed.
C) Do not delete Jacob that way his memory will not leak
D) Add another class that will handle all the memory for us.

Question #7

Assuming a destructor for the Person class is a solution to fix our memory leak problem. How can a destructor be properly setup in the previous Person class to fix this problem?

A)
```
~Person()
{
	std::cout << "Cleaning up memory" << std::endl;
}
```

B)
```
Person()
{
	std::cout << "Cleaning up memory" << std::endl;
	delete m_height;
}
```

C)
```
~Person()
{
	std::cout << "Cleaning up memory" << std::endl;
	delete[] m_height;
}
```

D)
```
~Person()
{
	std::cout << "Cleaning up memory" << std::endl;
	delete m_height;
}
```

Study Questions – Answers

Question #1 : B

- X is not a memory address, make this work as intended we'd have to provide an address to the pointer instead of just a number. To do this put a & operator in front of the variable.

Question #2 : A

- Whilst many of the other answers are true, they are only true in specific situations this answer will always be true. A pointer is just a variable that holds a Memory address location in your computer, it doesn't specifically have to be another variable's address within your program.

Question #3 : C

- Smart pointers act mostly the same as regular pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used. This is known as RAII – Random Acquisition is Initialization.

Question #4 : A

- *pointer + 2 is going to dereference (get the value of) the pointer at it's current memory address which is pointing to the first element in the array it is then going to take the value and add two to it giving us 17. A is the only answer with 17 as the first number, therefore we know it is A.

Question #5 : A

- When assigning an array to a pointer the pointer will point at the first element in the array, so in a sense the pointer is just a regular array, but a pointer will not always be an array.

Question #6 : A

- Whilst the last answer to this question is a good thing we could do, it's how Smart pointers were made it's not the easiest solution whereas with answer A all we have to do is add like 3 lines of code and we're good to go.

Question #7 : D

- D is the proper way to make a destructor and to free memory from within that destructor for our Person class. Answer choice C is very close, but it messes up with the delete keyword, m_height is not an array you should only use delete[] when freeing memory for a dynamically allocated array.