Chapter 7 - Pointers

CSE 2010
Week 11

Review - Defining Pointers

- Pointers are variables whose values are **memory addresses**.
- These memory addresses are the addresses of other variables or objects.

Declaring/Initializing pointers:

1. `datatype* pointerName = &existing variable/object;`

   a.
   ```
   int year = 2021;
   int* pYear = &year; //pYear = memory address of year
   ```

   b.
   ```
   double salary = 50000.00;
   double* pSalary = &salary;
   ```

   c.
   ```
   Pet pet1(“Obi”, “cat”, ‘M’, 18.0);
   Pet* p = &pet1;
   ```

   Note: the pointer datatype needs to match the type of the variable whose address it stores.

2. `datatype* pointerName = NULL;`

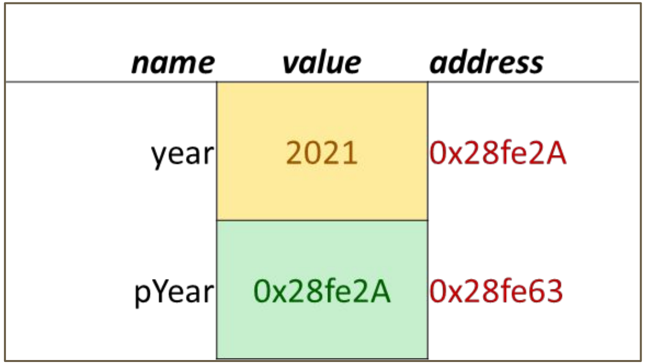
   a.
   ```
   double* pPrice = NULL;
   ```

   Note: Use this method when you need to declare a pointer but don’t yet know what it needs to point to.

Review - Dereferencing Pointers

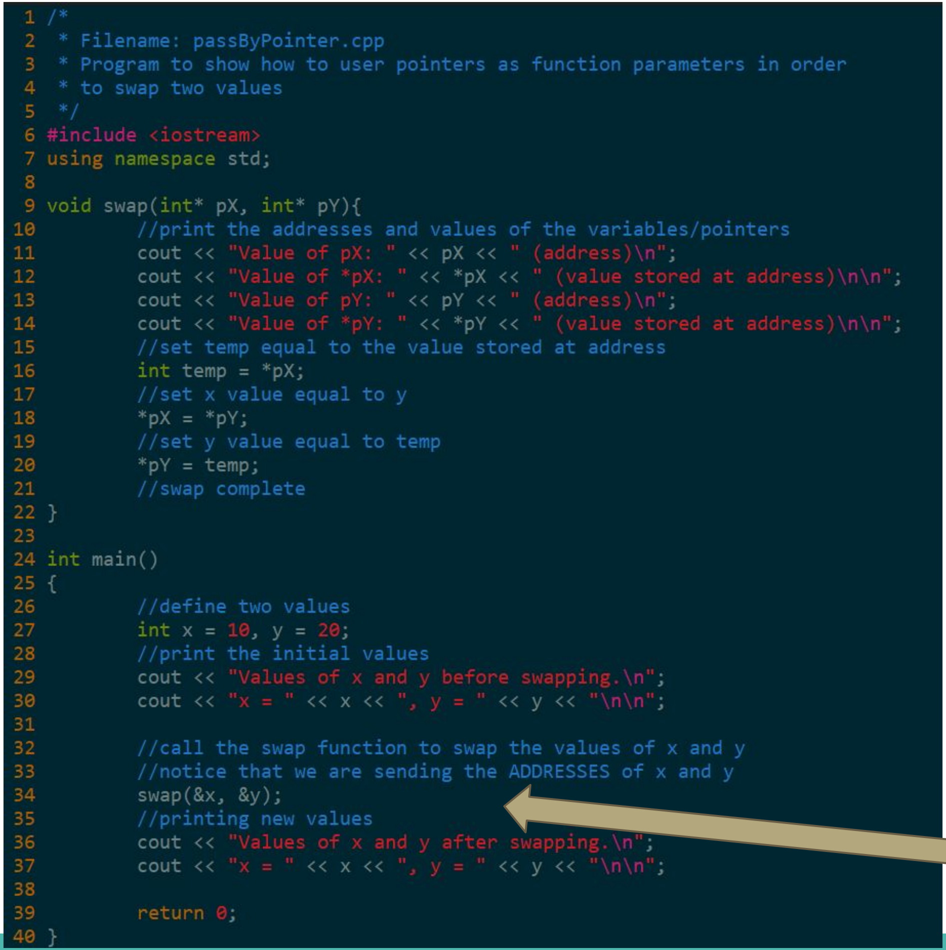
- We can use the dereference operator - * - to access or modify the value stored at the memory address a pointer points to.

```
int year = 2021;
int* pYear = &year;

cout << “Value of year: “ << year << “\n”; // 2021
cout << “Value of &year: “ << &year << “\n”; // 0x28fe2A
cout << “Value of pYear: “ << pYear << “\n”; //0x28fe2A
cout << “Value of *pYear: “ << *pYear << “\n”; // 2021
cout << “Value of &pYear: “ << &pYear << “\n”; // 0x28fe63
```



name	*value*	*address*
year	2021	0x28fe2A
pYear	0x28fe2A	0x28fe63

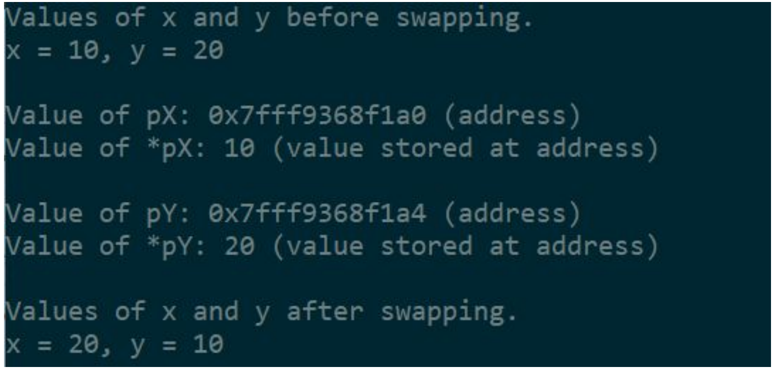
Review - Pass by Pointer - Swapping Value



```cpp
/*
 * Filename: passByPointer.cpp
 * Program to show how to user pointers as function parameters in order
 * to swap two values
 */
#include <iostream>
using namespace std;

void swap(int* pX, int* pY){
        //print the addresses and values of the variables/pointers
        cout << "Value of pX: " << pX << " (address)\n";
        cout << "Value of *pX: " << *pX << " (value stored at address)\n\n";
        cout << "Value of pY: " << pY << " (address)\n";
        cout << "Value of *pY: " << *pY << " (value stored at address)\n\n";
        //set temp equal to the value stored at address
        int temp = *pX;
        //set x value equal to y
        *pX = *pY;
        //set y value equal to temp
        *pY = temp;
        //swap complete
}

int main()
{
        //define two values
        int x = 10, y = 20;
        //print the initial values
        cout << "Values of x and y before swapping.\n";
        cout << "x = " << x << ", y = " << y << "\n\n";

        //call the swap function to swap the values of x and y
        //notice that we are sending the ADDRESSES of x and y
        swap(&x, &y);
        //printing new values
        cout << "Values of x and y after swapping.\n";
        cout << "x = " << x << ", y = " << y << "\n\n";

        return 0;
}
```

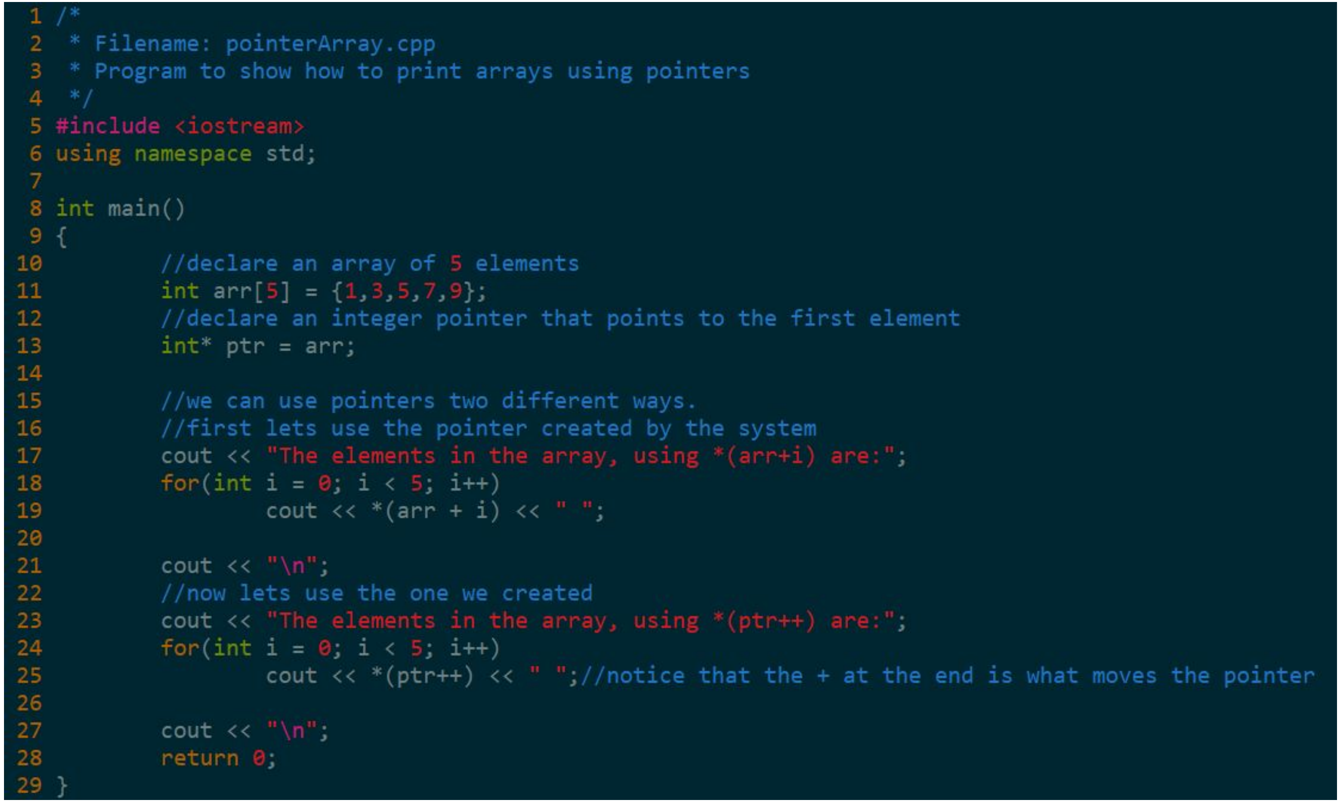
Key thing to remember is to pass the ADDRESS of a value

Output:



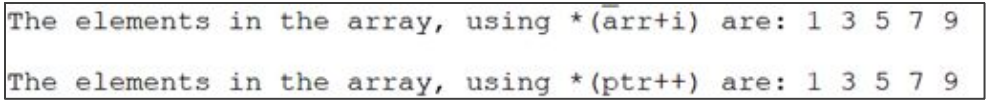
```
Values of x and y before swapping.
x = 10, y = 20

Value of pX: 0x7fff9368f1a0 (address)
Value of *pX: 10 (value stored at address)

Value of pY: 0x7fff9368f1a4 (address)
Value of *pY: 20 (value stored at address)

Values of x and y after swapping.
x = 20, y = 10
```

Review- Using Pointers to Access Arrays



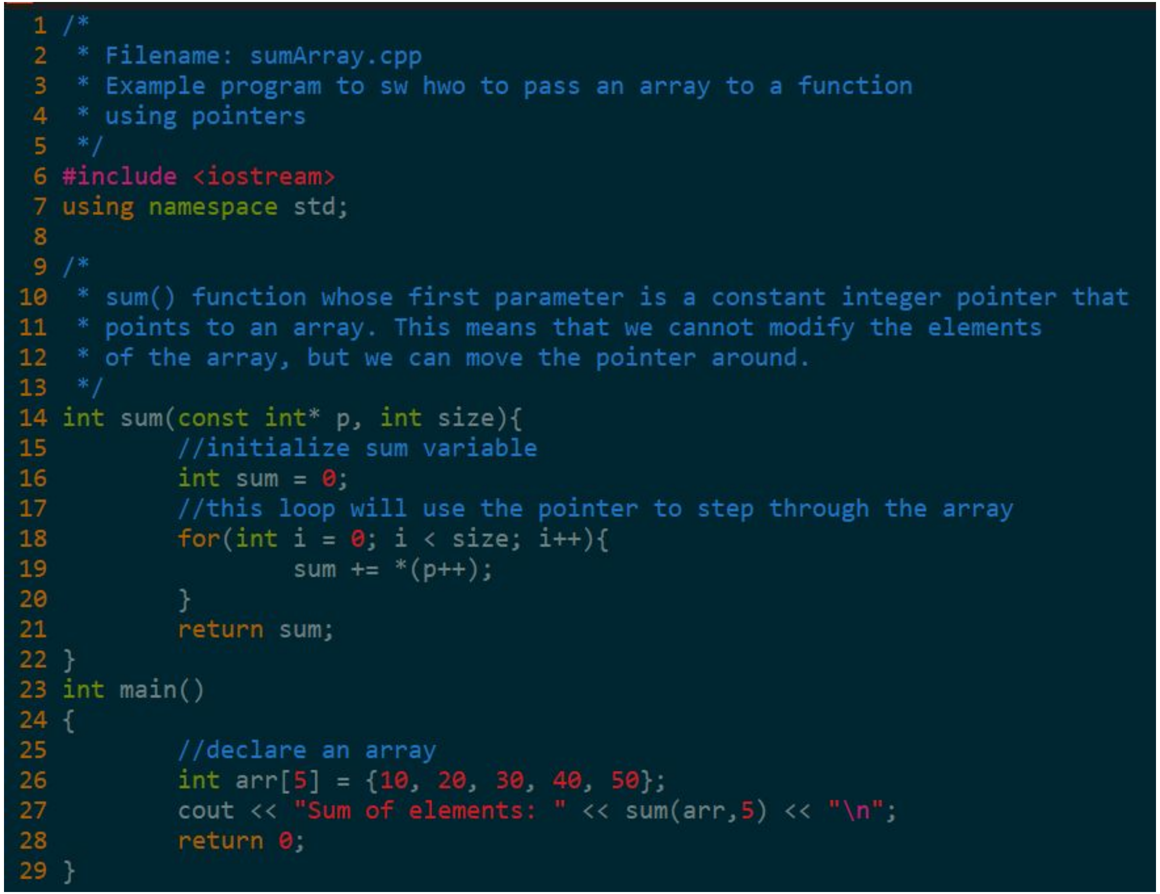
```cpp
/*
 * Filename: pointerArray.cpp
 * Program to show how to print arrays using pointers
 */
#include <iostream>
using namespace std;

int main()
{
        //declare an array of 5 elements
        int arr[5] = {1,3,5,7,9};
        //declare an integer pointer that points to the first element
        int* ptr = arr;

        //we can use pointers two different ways.
        //first lets use the pointer created by the system
        cout << "The elements in the array, using *(arr+i) are:";
        for(int i = 0; i < 5; i++)
                cout << *(arr + i) << " ";

        cout << "\n";
        //now lets use the one we created
        cout << "The elements in the array, using *(ptr++) are:";
        for(int i = 0; i < 5; i++)
                cout << *(ptr++) << " ";//notice that the + at the end is what moves the pointer

        cout << "\n";
        return 0;
}
```

Output:



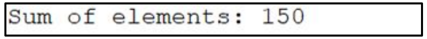
```
The elements in the array, using *(arr+i) are: 1 3 5 7 9

The elements in the array, using *(ptr++) are: 1 3 5 7 9
```

Review - Pass by Pointer - Passing Arrays



```cpp
/*
 * Filename: sumArray.cpp
 * Example program to sw hwo to pass an array to a function
 * using pointers
 */
#include <iostream>
using namespace std;

/*
 * sum() function whose first parameter is a constant integer pointer that
 * points to an array. This means that we cannot modify the elements
 * of the array, but we can move the pointer around.
 */
int sum(const int* p, int size){
        //initialize sum variable
        int sum = 0;
        //this loop will use the pointer to step through the array
        for(int i = 0; i < size; i++){
                sum += *(p++);
        }
        return sum;
}
int main()
{
        //declare an array
        int arr[5] = {10, 20, 30, 40, 50};
        cout << "Sum of elements: " << sum(arr,5) << "\n";
        return 0;
}
```

Output:



```
Sum of elements: 150
```

Let’s learn some more about pointers!

Let's talk about different types of memory...

Stack Memory

- Variables, global variables, objects, arrays, and vectors we have created are all stored in stack memory.
- Process of Stack Memory:
  - Stack memory is empty when a program starts.
  - As variables, objects, etc are defined in main(), values are stored onto the stack.
  - When functions are called, variables within the scope of the function are added to the stack.
  - Once the function ends, these variables are removed from the stack.
  - Stack memory is emptied when the final return statement is reached at the end of main().
- Stack memory is referred to as compile-time memory, because stack values are determined at compilation time.
- Very fast and efficient memory management

Heap Memory

- Stack memory is allocated during compilation time, but we can also allocate memory during run-time.
- There are times when we need to store very LARGE objects or collections of objects.
- Or there are times when we need to create objects that we want available globally, not to be deleted at the end of its scope.
- C++ environments reserve a large storage area called a **heap** to store objects created during run-time.
- Objects in heap memory do not have names associated with them, so we use pointers to access them.
- Pointers themselves are stored in stack memory, but heap memory can be used to store the object it is pointing to.

Using heap memory - `new` and `delete`

- So how do we actually create objects in the heap during run-time?
- Objects in the heap are not automatically allocated or deleted, so it is our job to explicitly do this.

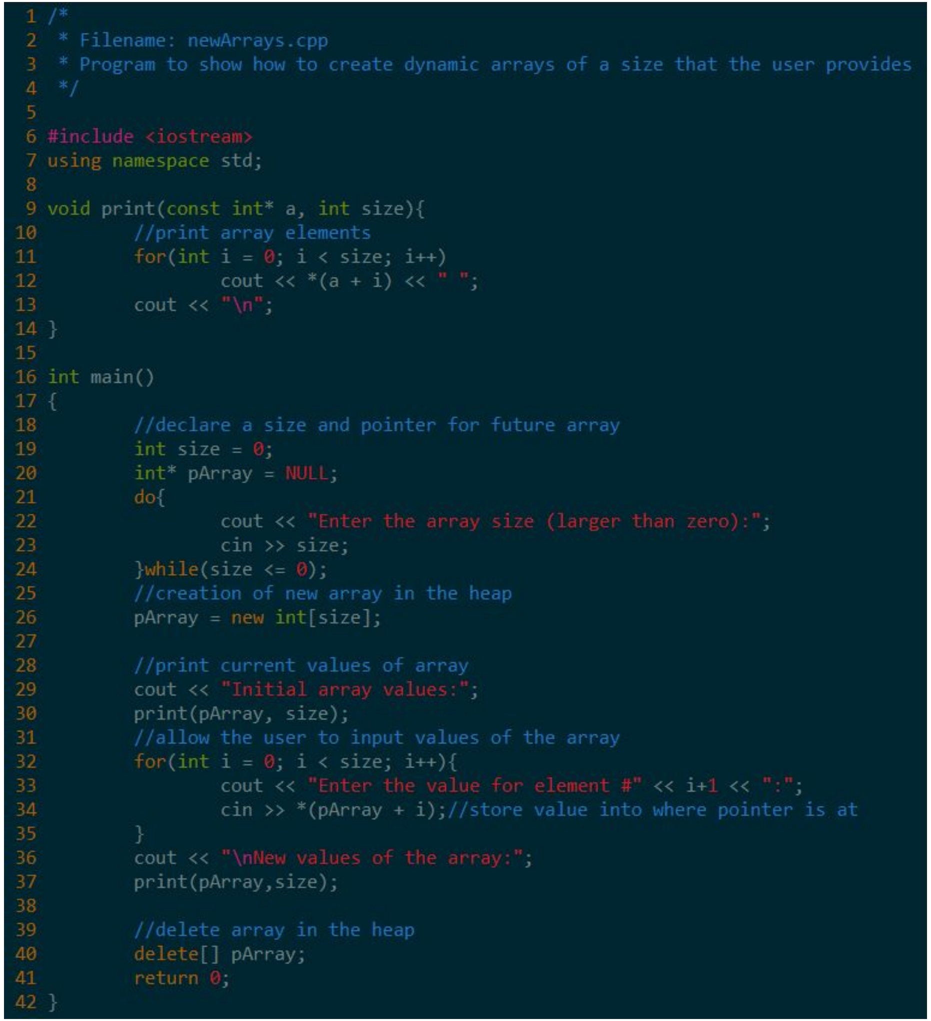
- We use the `new` and `delete` operators.
  - `new datatype` - used to create memory in the heap for a single object
  - `new datatype[size]` - used to create memory in the heap for an array of objects
  - `delete ptr`- used to delete the single object using its pointer
  - `delete[] ptr`- used to delete allocated memory of an array in the heap.

- It is VERY IMPORTANT to make sure that for every object you create with `new`, you delete that object before the program ends, or else that object will remain in heap memory.
  - Failing to do this will result in memory leaks and may cause your computer to act all weird.

Creating Arrays with the `new` operator

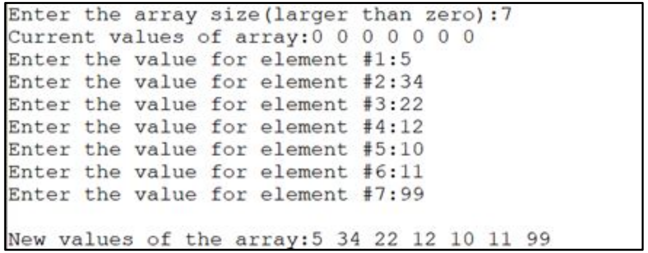
- So far, we learned that the arrays we created are “static” arrays whose size cannot be changed because it is determined at compilation time.
- Now we will learn how we can write a program that creates a variable-size array each time the user runs the program (dynamic arrays).
- Syntax:

```
datatype* pointerName = NULL;
//determine size of the array
pointerName = new datatype[size];
………//program runs
delete[] pointerName;
```



```cpp
/*
 * Filename: newArrays.cpp
 * Program to show how to create dynamic arrays of a size that the user provides
 */

#include <iostream>
using namespace std;

void print(const int* a, int size){
        //print array elements
        for(int i = 0; i < size; i++)
                cout << *(a + i) << " ";
        cout << "\n";
}

int main()
{
        //declare a size and pointer for future array
        int size = 0;
        int* pArray = NULL;
        do{
                cout << "Enter the array size (larger than zero):";
                cin >> size;
        }while(size <= 0);
        //creation of new array in the heap
        pArray = new int[size];

        //print current values of array
        cout << "Initial array values:";
        print(pArray, size);
        //allow the user to input values of the array
        for(int i = 0; i < size; i++){
                cout << "Enter the value for element #" << i+1 << ":";
                cin >> *(pArray + i);//store value into where pointer is at
        }
        cout << "\nNew values of the array:";
        print(pArray,size);

        //delete array in the heap
        delete[] pArray;
        return 0;
}
```

Output:



```
Enter the array size(larger than zero):7
Current values of array:0 0 0 0 0 0 0
Enter the value for element #1:5
Enter the value for element #2:34
Enter the value for element #3:22
Enter the value for element #4:12
Enter the value for element #5:10
Enter the value for element #6:11
Enter the value for element #7:99

New values of the array:5 34 22 12 10 11 99
```

Initializing Objects with `new`

- Say we have an Employee class, which creates object with a name and a salary.

  ```
  Employee* tina = new Employee(“Tester, Tina”, 50000.00);
  ```

  The above creates an Employee object using the overload constructor in HEAP memory.

- We can access the object with *

  ```
  cout << “Employee name: “ << (*tina).get_name();
  ```

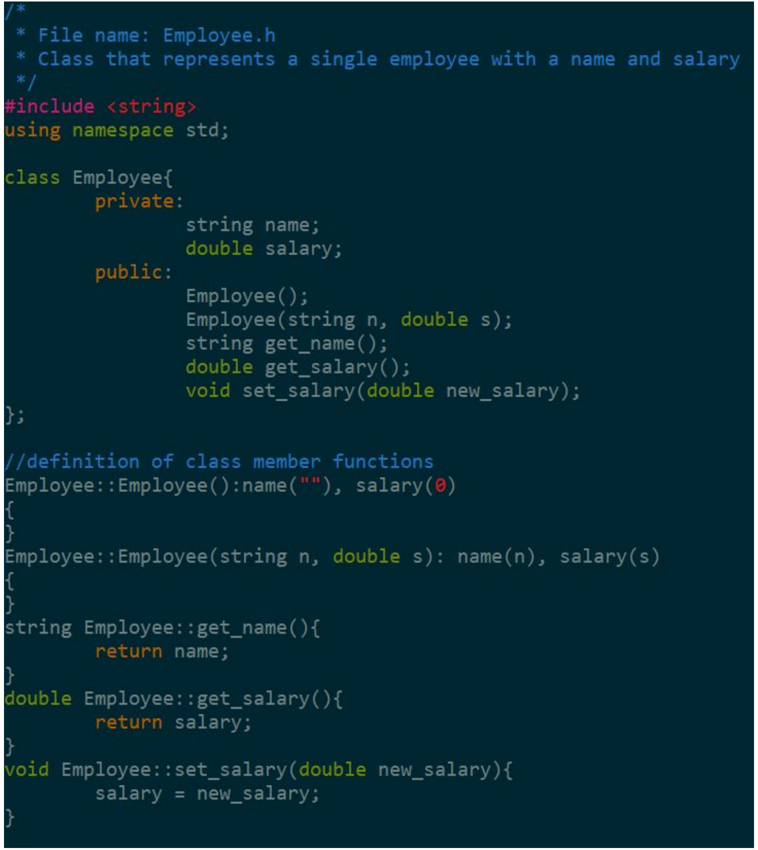
- As with other objects in heap memory, you will eventually have to delete the object.

  ```
  delete tina;
  ```

Sharing Values Between Classes with Pointers

- Consider the following Employee class.

- The Employee class represents an Employee with a name and a salary.



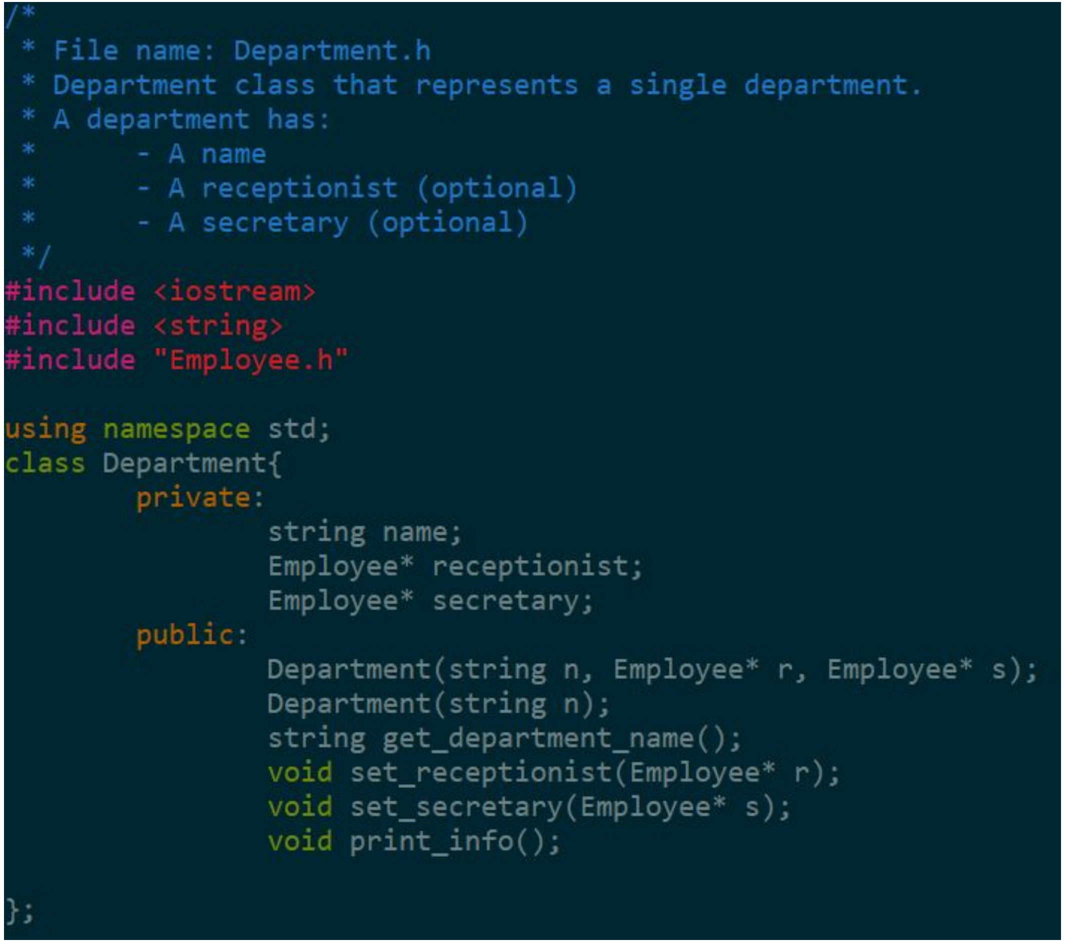
```
/*
 * File name: Employee.h
 * Class that represents a single employee with a name and salary
 */
#include <string>
using namespace std;

class Employee{
        private:
                string name;
                double salary;
        public:
                Employee();
                Employee(string n, double s);
                string get_name();
                double get_salary();
                void set_salary(double new_salary);
};

//definition of class member functions
Employee::Employee():name(""), salary(0)
{
}
Employee::Employee(string n, double s): name(n), salary(s)
{
}
string Employee::get_name(){
        return name;
}
double Employee::get_salary(){
        return salary;
}
void Employee::set_salary(double new_salary){
        salary = new_salary;
}
```

Sharing Values Between Classes with Pointers

- Consider the following Department Class (CSE, Physics, Math, etc)

- Each Department has a name, and optionally a receptionist and secretary (must have both).

- Receptionists and Secretaries are of type Employee

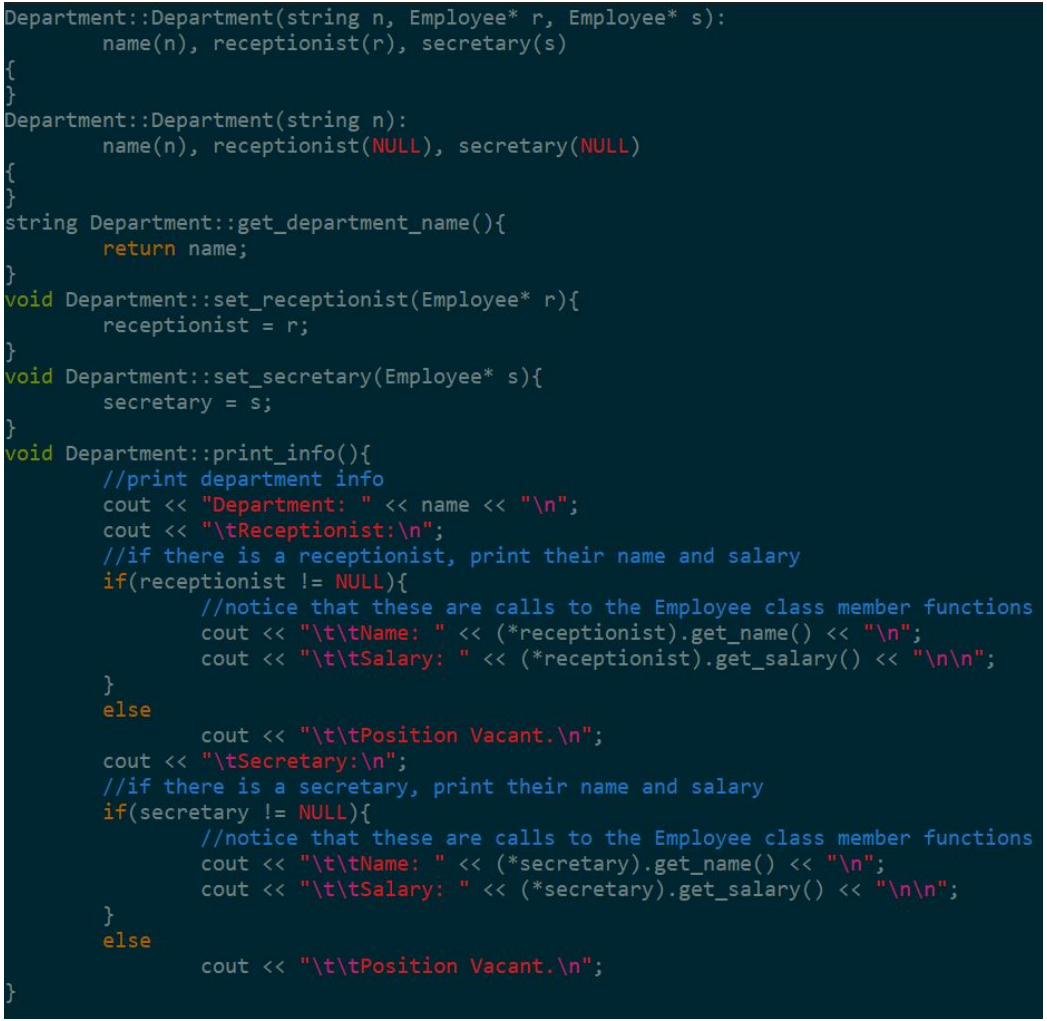


```cpp
/*
 * File name: Department.h
 * Department class that represents a single department.
 * A department has:
 *    - A name
 *    - A receptionist (optional)
 *    - A secretary (optional)
 */
#include <iostream>
#include <string>
#include "Employee.h"

using namespace std;
class Department{
    private:
        string name;
        Employee* receptionist;
        Employee* secretary;
    public:
        Department(string n, Employee* r, Employee* s);
        Department(string n);
        string get_department_name();
        void set_receptionist(Employee* r);
        void set_secretary(Employee* s);
        void print_info();

};
```

Sharing Values Between Classes with Pointers

- If a Department has a receptionist and secretary, then the pointer(s) will be set to the address of an already existing Employee object.

- If a Department does not have a receptionist and secretary, the the pointer(s) will be set to NULL.



```
Department::Department(string n, Employee* r, Employee* s):
        name(n), receptionist(r), secretary(s)
{
}
Department::Department(string n):
        name(n), receptionist(NULL), secretary(NULL)
{
}
string Department::get_department_name(){
        return name;
}
void Department::set_receptionist(Employee* r){
        receptionist = r;
}
void Department::set_secretary(Employee* s){
        secretary = s;
}
void Department::print_info(){
        //print department info
        cout << "Department: " << name << "\n";
        cout << "\tReceptionist:\n";
        //if there is a receptionist, print their name and salary
        if(receptionist != NULL){
                //notice that these are calls to the Employee class member functions
                cout << "\t\tName: " << (*receptionist).get_name() << "\n";
                cout << "\t\tSalary: " << (*receptionist).get_salary() << "\n\n";
        }
        else
                cout << "\t\tPosition Vacant.\n";
        cout << "\tSecretary:\n";
        //if there is a secretary, print their name and salary
        if(secretary != NULL){
                //notice that these are calls to the Employee class member functions
                cout << "\t\tName: " << (*secretary).get_name() << "\n";
                cout << "\t\tSalary: " << (*secretary).get_salary() << "\n\n";
        }
        else
                cout << "\t\tPosition Vacant.\n";
}
```

But couldn’t we make the Department class without pointers?

WITH pointers

```
class Department {
    ...
    private:
    string name;
    Employee* receptionist;
    Employee* secretary;
  };
```

Objects that don’t have a receptionist and secretary will not take up memory, since those attributes will be set to NULL.

WITHOUT pointers

```
class Department {
    ...
    private:
    string name;
    bool has_receptionist;
    Employee receptionist;
    bool has_secretary;
    Employee secretary;
  };
```

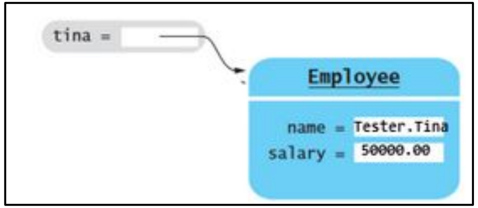
Objects that don’t have a receptionist and secretary will still take up memory and you need additional attributes.

In addition to using pointers within the class to share values, we can use pointers to create shareable objects in heap memory.

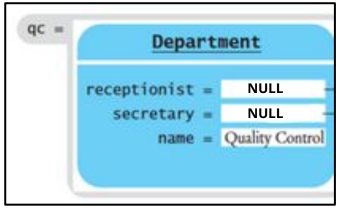
Sharing Objects - with Pointers

- Pointers enable us to properly share attributes between objects.

```
//declare an Employee pointer to initialize an Employee
//object in heap memory.
Employee* tina = new Employee(“Tester, Tina”, 50000.00);
```



```
//declare a Department object with no secretary/receptionist
Department qc(“Quality Control”);
```

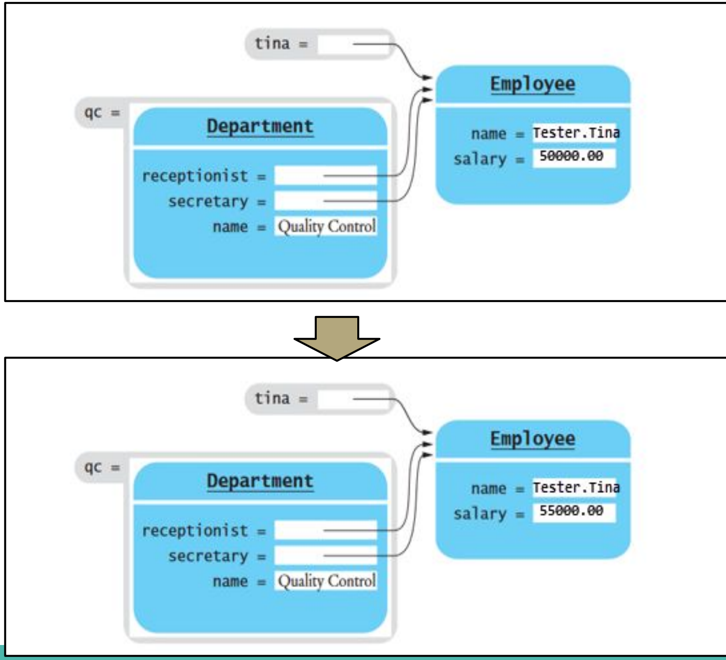


```
//set tina as the department’s receptionist/secretary
qc.set_receptionist(tina);
qc.set_secretary(tina);
```

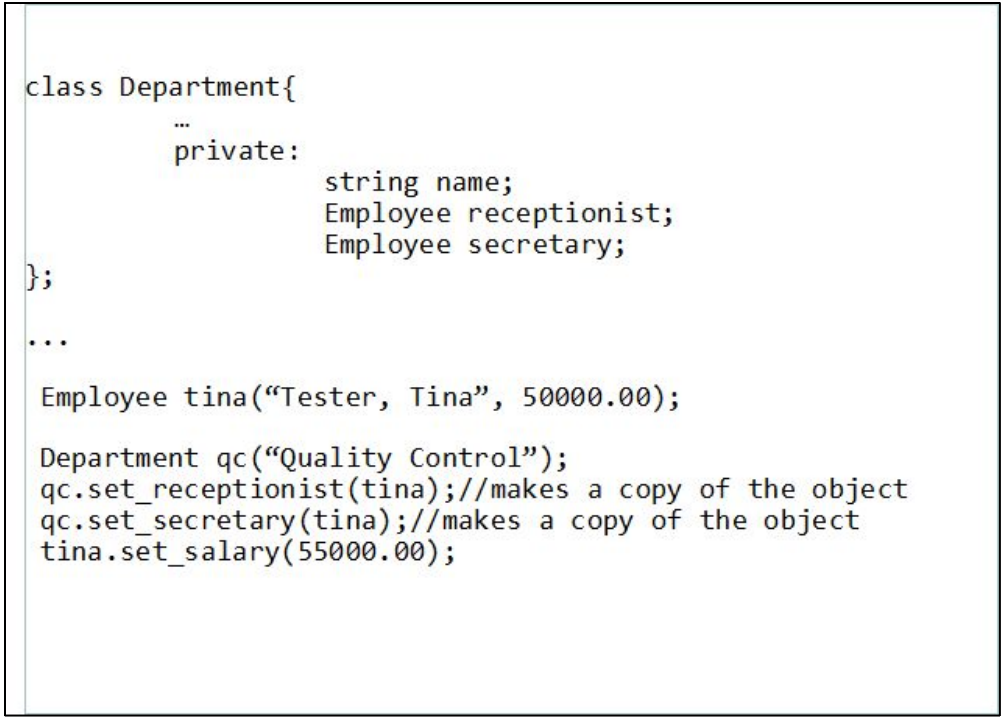
Sharing Objects - with Pointers

- Pointers enable us to properly share attributes between objects.

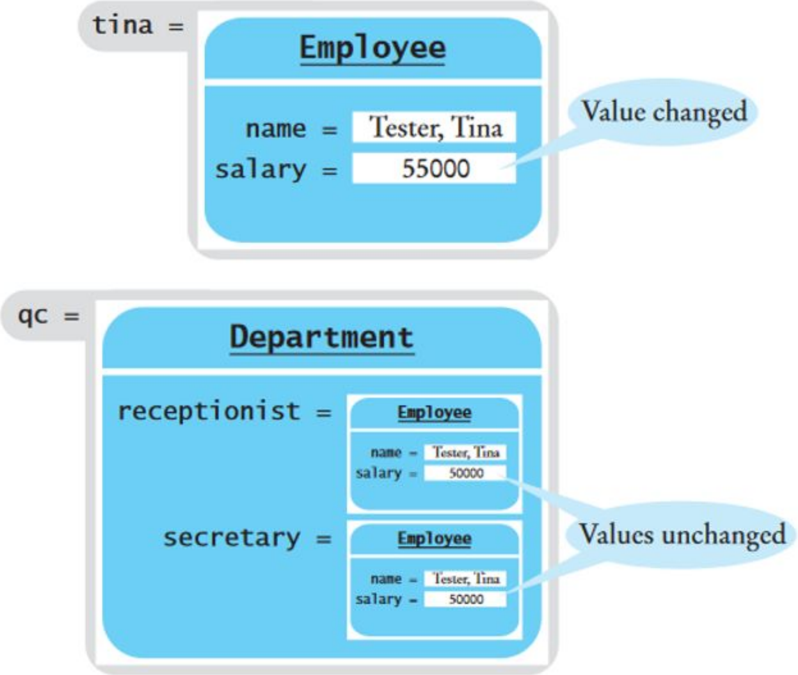
```
//declare an Employee pointer to initialize an Employee
//object in heap memory.
Employee* tina = new Employee(“Tester, Tina”, 50000.00);


//declare a Department object with no secretary/receptionist
Department qc(“Quality Control”);

//set tina as the department’s receptionist/secretary
qc.set_receptionist(tina);
qc.set_secretary(tina);


//update tina’s salary
(*tina).set_salary(55000.00);

delete tina;//once we’re done with program
```



What if we implemented this without Pointers...



```
class Department{
         …
         private:
                  string name;
                  Employee receptionist;
                  Employee secretary;
};

...

 Employee tina(“Tester, Tina”, 50000.00);

 Department qc(“Quality Control”);
 qc.set_receptionist(tina);//makes a copy of the object
 qc.set_secretary(tina);//makes a copy of the object
 tina.set_salary(55000.00);
```



Let’s look at a full example…. (zip folder with all files on Canvas)

Pointers Review

- In this chapter we learned how to use pointers to store memory addresses.
- Pointers can be used to indirectly access values through their addresses.
- We can use the address (&) operator to obtain addresses of variables.
- We can use the dereference operator (*) to access values stored at specific addresses.
- Pointers are helpful with the following:
  - Passing by Pointer
  - Accessing and modifying arrays
  - Using heap memory (new/delete)
  - Allows classes to have “optional attributes”
  - Sharing objects across classes.
- Pointers are also used in class inheritance...which we will see next week!