Final Exam Review

- Sadab Hafiz

Don’t take this as a single document to review for the exam. You should prepare for the exam properly by going through lecture slides and labs. If something is unclear, you can refer to the textbook or ask the UTAs and instructors.

What to review...

Pointers:

- Lab 9
- Pass by reference
- Pointers to objects

Dynamic Memory Allocation:

- Lab 9
- Dynamic Arrays

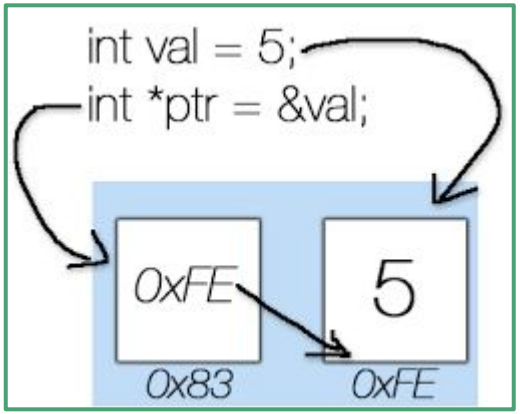
Class and Objects(OOP):

- Lab 10 and Lab 11
- Object Oriented Programming (constructors, getters, setters)

Recursion:

- Lab 13

Pointers



Pointer variables are able to store the memory addresses of other variables and objects used by the program. Knowing the address of an object can very convenient. You can freely pass it around into functions if the function needs to operate on that object.

Instead of referring to an entire object or a large variable, using pointers helps access much larger objects by passing it in a much smaller size.

&

695 Park Ave,
New York, NY 10065

*

695 Park Ave,
New York, NY 10065

Dereferencing

```
int main( ) {

      int var = 3;                          // initialize int variable var as 3

      int *p = &var;                        // pointer p points to the address of var

      cout << p << endl;                    // prints address of var (hexadecimal)

      int var2 = *p;                        // var2 dereferences pointer p

      cout << var2 << endl;                 // prints 3 because var2 is the value in &var

      int *p2 = p;                          // sets int pointer p2 equal to p

      cout << *p2 << endl;                  // dereference p2 and prints 3 (same as var)

      *p2 = *p + 7;                         // adds 7 to var (since p2 points to var)

      cout << *p2 << endl;                  // prints 10(7+3) by dereferencing p2

      cout << var2 << endl;                 // prints 3 (var2 is still 3)

      cout << *p << endl;                   // prints 10 (same as *p2 and var)

      cout << var << endl;                  // prints 10 (same as *p2 and *p)

}     // Given something like this, you should be able to tell the output
```

The -> operator

Using -> is an alternative to dereferencing objects of classes.

Dereferencing: We create a local copy so that we can access the data members of the object that a pointer is pointing to.

If you want to directly refer to the data member of the object, you can do it using the helpful operator ->. This way you don't have to create unnecessary local copies for every pointer object.

For example:
Imagine we have a Date class with data members month, year and day. Given a pointer variable to a Date object called d, we can access the day of the Date pointer using: d -> day

Automatic memory allocation

Normally, any variable “lives” only within the block where it is declared, and disappears once the program execution leaves this scope.

This memory management is called automatic, the program allocates memory for each variable when it enters the scope of the variable, and deletes that memory when leaving that scope.

This is how C++ handles local variables within functions and scope.

A problem with automatic memory allocation

The following function that's supposed to create a poem and return its memory address, will not work (reliably):

```
string * createAPoem() {

      string poem =    // making a string with a poem

      "   Said Hamlet to Ophelia,            \n"

      "   I'll draw a sketch of thee,        \n"

      "   What kind of pencil shall I use?   \n"

      "   2B or not 2B?                      \n";

    return &poem;    // and returning its address

}
```

Since the variable poem exists locally inside the function, after exiting the function, the memory allocated for this string gets claimed and freed.

Even though we returned the address where the poem was, after the function exits, that address may be taken and used by some other part of your program, the poem may be easily overwritten by some other value.

How do we fix this? Using keyword ‘new’

We can allocate a chunk of memory for the poem so that it would remain persistent and would not be claimed by the program after the function exits. Using the keyword `new`:

```
string * createAPoemDynamically() {

     string * ppoem;             // declare a pointer to string to store the address

     ppoem = new string;         // <-- DYNAMICALLY ALLOCATE MEMORY

                                 //     for a poem string and store its address in the pointer

     *ppoem =                    // put a poem there

     "   If you know              \n"

     "   you know                 \n";

     return ppoem;               // return the address where the poem is

}
```

More about keyword `new`

The address of a dynamically allocated memory can be passed around, returned from a function, or stored in another variable, etc. The dynamically allocated memory will remain persistently in the computer memory throughout the program execution:

```
int main() {

      string * p;

      p = createAPoemDynamically();

      // The memory at the address p still stores the poem we put in it during the function call. Neat!

      // At any later moment, you can print it out:

      cout << *p;

                              // You can also save the address into another pointer variable:

      string *p2 = p;         // then both pointers, p and p2, will be pointing to the same poem

      cout << *p2;

}
```

Cleaning up Dynamically Allocated Memory

Once a dynamically allocated memory is not needed, it must be manually released to the system, otherwise if we only keep allocating memory and never giving it back, we may run out of memory eventually. Clean up example:

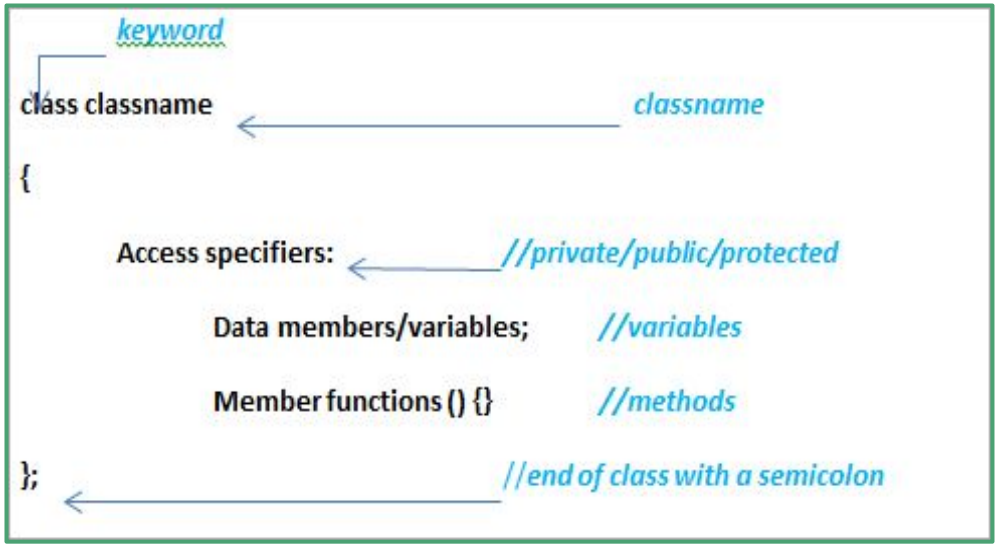
```
int *p = new int;                          // allocate an integer dynamically

*p = 1234;                                 // we are using it

cout << *p << endl;                        // still using it

// once it's not needed, delete it:

delete p;

// set the dangling pointer equal to nullptr

p = nullptr;
```

After deleting a dynamically allocated pointer, it is good practice to set the pointer equal to nullptr also known as **null pointer**. This is done to prevent any undefined behavior. Learn about dynamically allocated arrays from lab 9.

Classes Summary (recap)

In C++, classes are compound datatypes similar to struct, which contain:

- data fields also called data members
- member functions, which can operate on the data members
- constructors (special functions called when instances of a class get created)



Data members and member functions can be declared public, private or protected thus making them accessible or not accessible from outside the class. This principle is called **encapsulation**.

Access Specifiers

Access Specifiers are used to specify who can access certain data members and member functions of a class. There are three access specifiers:

public: Public member functions and data members can be accessed by anyone who has access to an object of this class including other classes and global functions. Usually there are getters and setter functions which allows users to modify private fields.

private: Private member functions and data members can be accessed ONLY by the class itself. Usually data members and sensitive functions are specified as private and so cannot be directly modified or accessed from outside the class.

protected: Share member functions and data members only with derived classes

You can find an entire class implemented in Lab 11. Take a look at the **Particle** class as an example of class declaration and implementation.

The “this” Pointer

The **this** pointer is used to refer to the current instance of the class. It is useful in cases where you need to refer to the data members of a class without being in conflict with function parameters.

For example, let’s say we have a Date class with data member month. If the setter function contains parameter same as the data member, we can do the following to implement the setter:

```
void Date : : setMonth(int month){

       // datamember = parameter

       this -> month = month;

}
```

Practice Questions about Pointers

You should be able to create the following:

- an uninitialized pointer
- a null pointer
- pointer to variables of different data types (int, float, double)
- pointer to class objects
- pointer to an array (click here to learn more)
- pointer in dynamic memory (using ‘new’ keyword)
- deallocating dynamic memory (using ‘delete’ keyword)
- fixing dangling pointers (after delete, set pointer to ‘nullptr’)
- dynamic array (click here for an example)

Practice Questions (definitions)

1. What is the difference between passing by value and passing by reference to a function?
2. Can functions return dynamic array? How?
3. In an array, what is the difference between the size and the capacity? Give an example.
4. How is a constructor function different from member functions? What is its purpose?
5. Why is the “delete” keyword used? Why is it important to set dangling pointers equal to “nullptr”?
6. What is the difference between .hpp and .cpp files?
7. Why is the “: :” operator used?

Nested Loops and Functions

You should be familiar with nested for-loops by now. Make sure you review and practice nested for loops. Look at prior labs.

You should also be familiar with functions. Review how to create functions. This includes functions that take pointer parameters or return pointers. Know how to pass arrays as parameters.

You should be familiar with reference(&). Passing by reference or using pointers means you can directly modify the variable or data being passed to the function.

You should know how to iterate through an array or a string.

Practice Question

What will be the output of the following?

```
int main() {

	int a = 20;

	int b = 30;

	int *c = &a;

	int *d = c;

	d = &b;

	b = *d + 10;

	a = *d - *c;

	cout << a << " " << b << endl;

}
```

Knowing the output is good. However, you should be able to tell exactly what is happening on each line and how the values of each variable are changing.

Practice Question about Class

Given a prompt, and some information about a class, you should be able to create a basic class. Try the following:

Create a class called Animal. The public interface should contain a name(string) , a pointer to it's direct ancestor(pointer to an Animal object), population(int). Use the main function to create two animals of your choice. Make one animal the ancestor of the other. Use nullptr as the ancestor's ancestor.

Similar to exam 1, there should be a cheat sheet provided. Use the cheat sheet to make sure your syntax is correct.

Another Practice Question about Class

Implement a class Rectangle. Provide a constructor to construct a rectangle given the height and the width. Add and implement two member functions get_perimeter() and get_area() to compute perimeter and area of the rectangle. Add a function resize() that takes two integers and updates the height and the width of the rectangle.

Bonus: Test all your functions in int main( )

Think about what should be private and public. Refer to Lab 11 if you're not sure. Logically what information should the user have access to?

Practice Question about 2D Arrays

Complete the following function that returns the average of all elements in a 2D array:

```
const int COL = 5;

int average( int values[ ][COL], int row) {

     // YOUR CODE GOES HERE

}
```

Here's a helpful website that covers how to pass 2D arrays to functions:
Passing two dimensional array to a C++ function

Practice Question about Dynamic Arrays

Use dynamic memory to initialize an int array with five rows with different lengths, and print it all out, so that it looks like this:

```
00000
1111
000
11
0
```

1. Allocate rows in dynamic memory and set the values appropriately
2. Print out all the rows
3. Clear the dynamic memory and fix dangling pointers

If you don't know how to do this, take a look at the lecture slides covering pointer arrays and dynamic memory allocation (Galton Board).

Recursion

Recursion is a programming technique where the solution to a problem depends on solutions to smaller instances of the same problem.

Programming languages support recursion by allowing functions to call themselves within their code.

For practice, solve at least the first four tasks in Lab 13. Understand the logic behind your solution. Try doing the bonus tasks for more practice.

Follow this link to learn more about recursion: Recursion Basics

Tips

- Go to the bathroom before the exam starts
- Study beforehand and get some sleep before the exam
- Use the provided cheat sheet to verify syntax
- Leave a bit earlier than usual so you don’t end up late
- Raise your hand during the exam if something is unclear
- If you see any major typo, raise your hand and ask a proctor
- If you are stuck at on question, move on and come back to it later

Good Luck!