I seperated this from the C++ document as it's a big enough topic on its own.

1 Intro to classes

- Big part of OOP style.
- A class differs from a structure, but it has similar syntax. A class's functions are private by default, but a struct's functions and members default to public.

```
//student.cc
struct Student {
    int assign, mt, final;
    float grade() {
        return assigns * 0.4 + mt * 0.2 + final * 0.4;
    }
};

int main() {
    Student s{60, 70, 80};
    Student s2{50, 0, 20}; //Me in 246
    cout << s.grade() << endl; //Prints out the grade of s
    cout << s2.grade() << endl; //Prints out the grade of s2
}
```

- We note that when we call the "grade" function, we are not passing any parameters.
- Within a class, the functions within the structure are called "methods" or "member functions" to specify it is within the structure.
- Methods take a hidden extra parameter called "this", which is a pointer to the object upon which the method is invoked. This is kinda like Java.
- Basically, the above code is equal to really saying:

```
float grade(this) {
    return this->assigns ...; //this is a pointer to the Student object.
}
\item IOW, ``this'' = ''&s'' in this example.
```

2 Initializing objects

- By default, we can easily initalize a struct-class like:

```
Student billy{60,70,80}; //This fills it in the order of the structure.
```

- If we want to use less values or something, we would have to overload our structure initalization. For example:

```
struct Student {
    //...
    Student (int assns, int mt, int final) {
        this->assns = assns;
        this->mt = mt;
        this->Final = final;
    }
};

//...
Student billy = Student{60, 70, 80};
```

- We MUST notice that we cannot do a default call anymore! That is, we cannot do something like "Student s3", as it is expecting three values.
- Once we define a constructor (that is, the function has the same name as the struct name), WE CANNOT USE THE DEFAULT CONSTRUCTOR.
- In C, we could initalize an integer with "int x(5)", or a string as "string s("hello")". Using or () are both valid, though in C++, we prefer syntatically.
- What if we write something like:

```
Student *pJane = new Student{70, 80, 90};
```

- pJane would be on the stack, but the value of that Student object is on the heap.
- What are the advantages of creating our own constructors (ctor)?
  - We can set default parameters, just as in functions, for our objects.
  - We can use overloading. Once again, like functions.
  - We can easily add sanity checks within the constructor, to accept or reject certain object initalizations.
  - We can add logic that alters from the default ctor.
- With default parameters, we could do something like:

```
struct Student {
    //...
    Student (int assns = 0, int mt = 0, int final = 0) {
        this->assns = assns;
        this->mt = mt;
        this->Final = final;
    }
};
```

```
//...
Student mary {70, 80}; //Defualts to 70, 80, 0
Student john; //Defaults to 0, 0, 0
```

- Consider the following example:

```
int z;
struct MyStruct {
    const int myConst = 5; //MUST BE INITIALIZED
    int &myRef = z; //MUST BE INITIALIZED
};
```

- In the above example, all of them will have the exact same default values of myConst = 5, and &myRef = z.
- How can we solve this if we want different values? The issue is that constants and references MUST be initalized.
- We couldn't do it in a ctor body, as by definition, fields must be constructed by the time the ctor body runs.
- We have to understand what happens when an object is created:
  - Space is allocated in memory.
  - Fields are constructed.
  - Ctor body is run.
- So we see that we MUST initalize our constant fields at the second step.
- The solution is to use Member Initialization List (MIL). This is executed before the constructor runs.
- We show an example w/ our Student class:

```
struct Student {
    const int id;
    Student (int id, int assns, int mt, int final) :
        id{id}, assns{assns}, mt{mt}. final{final} //id -> field, {id} ->
    {
        //ctor body...
    }
};
```

- You could use this.idid, but apparently it doesn't matter.
- As usual, note we can use or (). Makes no difference.

- Since the initialization of the fields happens BEFORE the ctor, it's part of the second step, so it's fine.
- Some things we should note about MIL:
  - We should always use MIL to initialize fields (not just constants and references), though it isn't really mandatory if not using constants or references (note if using const/references, you MUST use MIL).
  - Another thin to note is that fields in MIL are initialized in the order that they are declared in the class. Thus, this may affect some things if you require prior variable initializations first.
  - MIL can be used more efficiently than initialization within the body of the ctor. This is as you are initializing the field during construction, as opposed to declaring and then initializing. I think.
  - For example:

```
struct Student {
    string name;
    //...
    Student (const string &name) {
        this->name = name; //Constructed, default initialized, then i
    }
};
```

  - We want to avoid the above. This is slow. MIL is better.
- Consider this example:

```
struct Vec {
    int x = 0, y = 0;
    Vec (int x): x{x}
    {...}
};
```

- What happens? Is x=0 or x?
- In C++, MIL will take precedence, so x = x. This is bad style, but good to know.
- Consider the following bit of code, presuming the Student class was called:

```
Student billy{60, 70, 80};
Student bobby = billy; //What happens?
```

- We note that bobby will not call the initialization constructor, but rather the default copy constructor.
- The values of billy are copied to bobby to be initialized.
- We can, of course, override the default:

```
struct Student {
    //...
    Student () {...}
    Student (const Student &other) {
        assns{other.assns}, mt{other.mt{, final{other.final} { //Default
            //...
        }
}
```

- We see it's literally just another constructor that takes in an object as its parameter.
- We note that every class will come with six default methods:
  - normal ctor
  - copy ctor
  - copy assignment operator
  - destructor
  - a move ctor
  - a move assignment operator
- Of course, we can override every single one.
- We have to be careful when working with classes. For example, consider the following code:

```
struct Node {
    int data;
    Node *next;
    Node (int data, Node *next): data{data}, next{next} {
    }
};

Node *n = new Node {1, newNode{2, newNode{3, nullptr}}}; //uses normal ct
Node m = *n;  //uses copy ctor
Node *p = new Node {*n};  //uses copy ctor
```

- Do note that n is on the heap, pointing from 1 -¿ 2 -¿ 3, in the heap.
- m is on stack, pointing from 1 -¿ the 2 from n. This is a shallow copy.
- p is a 1 pointing to the values of 2 and 3, but on heap. This is also a shallow copy.
- A shallow copy is that there are shared values. The '1' is different in each case, but the 2 and 3 are what n was using!

- We WANT to be on the heap. This is why we would want to avoid such behaviour if, say, we want to avoid shallow copying.
- We can kinda avoid this by defining our own copy ctor:

```
struct Node {
    //...
    Node (const Node &other): data{other.data}, next{other.next? new Node
    }
};
```

- IOW, we are recursively creating new Node objects to link our nodes to in the heap. This means nothing is shared!
- We call the copy ctor when:
  - An object is initialized by another object.
  - When an object is passed by value.
  - When an object is returned by a function.
- We must be careful with ctors that can take in one parameter. This isn't saying to not use them, but there is the potential risk of accidental use:

```
struct Node {
    //...
    Node (int data) : data{data}, next{nullptr} {
    }
};

Node n(4); //Fine
Node n = 4; //Fine
int f (Node n) { ... }
f(4); //Fine
```

- We can fis this with the "explicit" keyword:

```
struct Node {
    //...
    explicit Node (int data) : data{data}, next{nullptr} {
    }
};

Node n(4); //Fine
Node n = 4; //Not fine
int f (Node n) { //... }
f(4); //Not fine
```

- Recall that with the default constructor, we can pass zero arguments.

- Thus, if we are planning to override the default constructor, we should REALLY make sure that we are able to accept zero arguments. We can do this with default parameters.
- We use destructors when we destroy an object. A method destructor (dtor) runs:
  - dtor body runs.
  - dtors are invoked for fields that are objects, in reverse declaration order.
  - space is deallocated.
- Syntax for destructor: Called with “delete np;”.
- An example:

```
~Node() {
    delete node;
}
```

- Now what if we create a Node, like “Node n 1, new Node 2, new Node 3, nullptr;”? What is invoked?
- We see that we invoke the default ctor.
- If we say “Node m = n;”, we are using the copy ctor.
- If we say “Node w;”, we are using the default built in ctor.
- But what if we say ”w = n;”? What are we doing? Well, we’re assigning, so we invoke the assignment operator.
- We can overload the copy assignment operator as well. For example:

```
Node &operator=(const Node &other) {
    data = other.data;
    delete next; //Remove the next value of this.
    next = other.next? new Node (*other.next) : nullptr;
    return *this;
}
```

- Something we should note is that something like this is perfectly valid.

```
int x = 5;
x = 10;
x = x;
```

- But our current overloaded operator does not support this! We must fix this:

```
Node &operator=(const Node &other) {
    if (this == &other) return *this;
    data = other.data;
    delete next;
    next = other.next? new Node (*other.next) : nullptr;
    return *this;
}
```

- Let us say we want to make a method that allows us to SWAP the values of nodes. We do the following, then use the swap function to rewrite the assignment operator:

```
void swap (Node &other) {
    swap(data, other.data);
    swap (next, other.next);
    //This will ensure there is no local copy.
}

Node &operator=(const Node &operator) {
    Node tmp = other;
    swap(tmp);
    return *this;
}
```

- Consider "Node m = plusOne(n);". This seemingly won't work, as plusOne(n) is an rvalue, and other is an lvalue reference. But plusOne(n) will create a temp value, so it works.
- But this could be wasteful, as we have to use temp memory to store intermediate results.
- We would try to store this value instead for use. But then we're working with an rvalue again!
- We use the move ctor, which takes an rvalue, allowing us to avoid storing an intermediate temp value, and instead directly work with the rvalue. IOW, we want to make plusOne(n) to immediately store its result into n, instead of saving, copying, then deleting the temp value.
- "x" is an RVALUE REFERENCE.
- An example:

```
Node (Node &&other):
    data{other.data}, next{other.next} {
        other.next = nullptr;  //Point back to nullptr.
    }
```

- We never create a new value or copy. Instead, we point directly to the result using MIL.
- "Node m4; m4 = plusOne(n);" will call the assignment ctor, so we modify that as well:

```
Node &operator=(Node &&other) {
    using std::swap;
    swap(data, other.data);
    swap(next, other.next);
    return *this;
```

- We see that the "other" value is deleted once we go out of scope! We "delete" the values of other this way, w/o an explicit delete.
- But as we already swapped values, we don't care.
- This is much more efficient than duplicating the value into temp, swapping, then deleting.
- "We kill him after we bring him to our office." - Victoria
- A tidbit: We can do "int x = y = z = 0;". This is because we do "z = 0", where the assignment operator returns the value of z, which is 0. Repeat. We must make our assignment operator to do the same.
- Recall the vector structure definition. Let us consider a function, "makeAVec":

```
struct Vec {
    int x, y;
    //...
    //ctor
    //copy ctor
    //move ctor
};

vec makeAVec() {
    return {0, 0};
}

int main() {
    Vec v = makeAVec();
} {
    return {0, 0};
}

int main() {
    Vec v = makeAVec();
}
```

- What happens at main?
- Logically, the compiler will try to invoke the move ctor if defined. If not, it will try to invoke the copy ctor.
- But actually, it will invoke the copy/move elision. The compiler realizes that what we need to do is initialize v. We don't need to copy or move. The compiler may try to optimize to reduce the number of copies required, and instead only calls the ctor instead of move or copy ctors.
- You are not expected to know when the compiler uses the elision optimization technique. The issue with it is that the behaviour completely depends on the compiler and what flags are set. It is thus hard to determine the expected behaviour of the program unless you explicitly know that the flag for disabling elision is set (in which you can assume the copy/move ctors will be called).
- Use the "-fno-elide-constructors" to stop copy elision. But this is not recommended.
- We use "The Rule of Five". That is, if we have to create ONE ctor, we have to create all five ctors (for example, to deal with shallow copies).
- Consider the vector object:

```
struct Vec {
    int x, y;
    Vec operator+(const Vec &other) {
        return {x + other.x, y + other.y};
        //other is always the second one, this is always the leftmost (an
    }
    Vec operator*(const int k){
        return {x*k, y*k};
        //ONLY works with v1 * int.  This is as the operator needs to see
    }
};
//Standalone functions:
Vec operator *(const int k, const Vec &v) {
    return v * k;
    //Force it to be called in the right order.  This works as its overlo
    //This MUST be a seperate function, as our left side is NOT a vector!
}

//Something like overriding the << and >> operators would also be standalo
//able to put anything but vectors on the left side.
```

- We note that there are some operators that are ALWAYS going to be defined as methods/members, and not functions. These are:
  - operator= (assignment)

  - operator[] (arrays)
  - operator-¿ (define member access through pointers)
  - operator()
  - operatorT

- We should seperate compilabion for classes. For example:

```
//node.h:
#ifndef NODE.H //Prevents duplicate declarations.
#define NODE.H
struct Node {
    int data;
    Node *next;
    explicit Node (int data, Node *next = nullptr); //explicit is used to
    Node(const Node &n);
};
#endif

//node.cc:
#include "node.h"
Node::Node(int data, Node *next = nullptr): //... {}
//:: refers to scope resolution operator.  That is, we are defining metho
Node::Node(const Node &n): //... {}
```

- Let us consider the vector structure again:

```
struct Vec {
    int x, y;
    //Vec (int x, int y) : x{x}, y{y} { //WRONG, SEE BELOW
    Vec (int x = 0, int y = 0): x{x}, y{y} {
        //...
    }
};

int main() {
    //REMEMBER that once we define our own constructor, we cannot use the
    Vec *vp = new Vec[10]; //pointer to Vec array of size 10 using defaul
    Vec moreVectors[15]; //Vec array of size 15 using ctor call with no a
    //Or if we don't want to fix the structure itself:

    Vec **vp2 = new Vec *[3]; //heap array, must manually initialize.
    vp2[0] = new Vec{0,0};
    vp2[1] = new Vec{1,1};
    vp2[2] = new Vec{2,2};
    Vec moreVectors[3] = {Vec{0,0}, Vec{1,1}, Vec{2,2}}; //stack array, m
```

```
    //For vp2, we would have to manually release all the heap values first
    for (int i = 0; i < 3; ++i) {
        delete vp2[i];
    }
}
```

- If we want to make sure a function doesn't modify any fields within an object, then in the function (both .h and .cc), put "const" after the function identifier.
- But what if we want to modify only one value within our class? For example, counting the number of times the value is called. We could fix this by removing const, but what if we want to use both const and be able to modify?
- Well, we see that something like the number of times a value is called is really irrelevant to the object itself, and not really that related.
- What we can do is declare a specific field to be mutable. Simply put "mutable" before the variable's declaration. This allows constant functions to be able to mutate that field:

```
//.h
struct Student {
    int assns, mt, final;
    mutable int numMethodCalls = 0;
    float grade() const;
}

//.cc
float Student::grade()const {
    ++numMethodCalls;
    return assns * 0.4;
}
```

- What if we wanted to count the OVERALL times a method is called across ALL of the objects? That is, the number of instances of that class.
- The "static" keyword allows for a value to not change throughout object creations.
- Without it, each object would have seperate fields that increment. This would be stuck at, say, 1, depending on how you implement it.
- Instead, use "static' before the variable's declaration. The field should be initialized in the implementation file, as the static field is associated with the class. They should only be declared in the interface file:

```
//.h
struct Student {
    int assns, mt, final;
    mutable int numMethodCalls = 0;
    float grade() const;
    static int numInstances;
}

//.cc
int Student::numInstances = 0;
static void printNumInstances() {
    cout << numInstances << endl;
}
```

- Mutable fields can be changed, even if the object is constant.
- Static fields are associated with the class itself, and not with a particular object.
- Static fields must be defined outside the class/structure! That is, it cannot be defined inside the interface, but it can be declared in the implementation.
- A static method does not depend on a specific object, and does not have an implicit "this". They can only access static fields. See the earlier snippet.
- MIDTERM CUTOFF. ==========
- Consider the example:

```
struct Node {
}

int main() {
    Node n{1, newNode {2, nullptr}};
    Node n2{3, nullptr};
    Node n3{4, &n2};
}
```

- This has a problem, as the stack will delete n2. But n3 has a pointer to n2!
- Remember nullptr is a value on the heap. But once we hit the end of stack, we call destructors. We delete n1, n2, but the "next" value of n3, which is either nullptr or a value on the heap, is also deleted! Thus, n3 will throw an error upon the stack ending.
- An "invariant" term is one that we rely on to be true during execution. In this case, we rely on Node having a nullptr or heap value on its next value. It's very hard for a programmer to ensure that something that we want to be invariant STAYS invariant when the user is using the code.

- We are to try to define invariance and to protect the user from harming themselves. IOW, we have to enforce invariance.i
- Our mistake earlier was that we allowed the user to access the fields in general.
- We can do so by making our fields private. But by default, for structs, they are all public, and you have to switch them to private with the "private" keyword.
- But another way to do it? Swap "struct" with "class". This will make it so that all fields are by default private, and you have to decide explicitly what fields are public:

```
class Node {
    //...
    public:
        //...
    private:
        //...
}
```

- Note all previous things we did apply to classes using the class keyword, so nothing else really changes.
- So consider the previous problem with the Node. We can instead use a wrapper to make the ENTIRE Node class private, and ensure only our functions are used to modify the node list.

```
//.h
class List {
    struct Node; //Remember, default private
    Node *theList = nullptr;
    public:
        void addToFront(int n);
        int ith (int i);
        ~List();
};

//.cc
//We place Node in the implementation as we don't want the client to see
//Techincally class or struct doesn't matter as it's already private and
#include "... .h"

struct List::Node { //Remember to add the List:: part!
    int data;
    Node* next;
    Node (int data, Node *next): data{data}, next{next} {}
    ~Node() {delete next;}
};
```

```
void List::addToFront(int n) {
    theList = new Node (n, theList);
}

int List::ith (int i) {
    Node *cur = theList;
    for (int j = 0; j < i && cur; ++j, cur = cur->next);
    return cur->data;
}

List::~List() {
    delete theList;
}
```

- We can say that Node is a "nested" class, that is by default also private. Thus, the user can only access Nodes through Lists. And you can only modify the list through the public functions.
- This allows us to control the user and ensure they don't do something stupid. In other words, we enforced invariance, as they cannot access the "next" field in Node. Instead, we force it to point to the nullptr at first, and then the heap. This thus works!
- Note that this is O(n^2).$This is a very, VERY slow method$!

¡++¿