CS 246 NOTES

Notes typeset and compiled by Joey Pereira

Disclaimer: These notes are not meant to be a replacement for lecture. They are meant to be a method of learning for myself for LaTeX, and are meant to be used for reference or review of content. I share these in hopes that it helps others understand or review the content covered.

Contents

Chapter 1

Introduction to Linux

1.1 History of the Shell

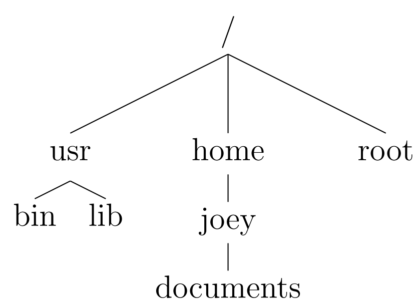
- Unix Shell (made in the 70's) called "shell" written by Bourne
- Other random shells made at the time, csh (c shell), ksh (korn shell)
- csh was revised into turbo shell3
- sh was named Bourne shell after many shells began appearing
- Bourne Again shell (bash) was created as a replacement for Bourne Shell

1.2 Linux File System

Composed of

- files (programs, data)
- directories (files/directories)

Directories form a tree-like structure



The initial or *root directory* of the tree or filesystem is known as "/" Absolute paths to a file in the file system always start with the root directory
Example: "/home/joey/document.txt", "/usr/bin/firefox"
The *Current directory* is directory you are currently sitting in, which you can see by running

```
$ pwd
```

which displays the absolute directory currently. *Relative paths* don't start with a "/". Examples: "joey/documents", "./firefox"

1.2.1 Special Directories

- "." → current directory
- ".." → refers to parent of current directory
- "~" → your home directory, also you can just do "cd" to get to home
- "~userid" → is that users home directory

1.3 Linux Programs

Where ever you are while in directories, you can run programs via the command line.

- does not show any file starting with a dot
- using "ls -a" will list out all of the files including hidden ones
- Wildcard matching:
  aside: in linux there is no enforcement of what the extension means for a file e.g. binaries could be .txt, no special meaning
  example: wanting to list files that end in .txt in the current directory "ls *.txt", "*.txt" is called a globbing pattern
  Shell substitues *.txt with every file in the current directory that matches the pattern onto the command line

echo ⟹ simply echos whatever you type back
rm ⟹ removes (deletes) files
cat ⟹ concatenates the files listed

Ctrl + C to kill a process
note that when we do cat it just repeats what we type
It would be useful if we can capture it
done by cat > nameofafile.txt
But we need to do a clean exit, so Ctrl + D does an EOF to thevim program
Output redirection
Using < and also > and whatnot
Every program has 3 streams
Standard input, standard output and standard error

The standard output and error defaults to stream
Default input defaults to linked to keyboard
To redirect error e.g. program.exe 2> error.log

Permissions: 3 groups of 3 bits example: rwx r-x r– first 3 are what the owner can do second iwhat members of the group (except owner) can do other bits what all can do Inside each group we have a read bit, a write bit, and an execute bit. If the bit/permission is not set, it appears as a dash.
Permissions: for files read - read the content of the file write - modify the content of the file execute - execute the file as a program
for folders read - list the contents of the folder, globbing, tab completion write - add or remove files in the directory execute - directory can be navigated to e.g. cd
Changing permissions only the owner can change permissions chmod mode file mode ownership-class operators permissions u-user + add r g-group - remove w o-other = set permission exactly x a-all
example: give others the ability to read permissions chmod o+r file revoke execute permissions from group chmod g-x file make everyones permissions rx chmod a=rx file
scripts are text files containing a sequence of linux commands execured as a program example: print the date, current user, current directory # !/bin/bash date whoami pwd
if current directory i not in $PATH, run script as ./script
redirectoing to /dev/null will delete any buffer (black whole)
programs return a status code 0 - successfull non-zero - failure $? is set to the status code $0 is the name of the currently running file
Looping over a list ex. REname all .cpp files to .cc #/bin/bash for name in *.cpp; do mv $name $name%cppcc done
ex. How many times does $1 occur in $2 (file) # !/bin/bash x=0 for word in ‘cat $2‘; do if [$word = $1]; then $x=$(x+1) fi done echo $x
Good idea to put input variables in double quotes to code defensively so space delimited input is never executed
Testing - no one likes testing - create test suite before actual code Functional Testing - run every function at least once - within a function - cover different control flow - code coverage - variable ranges: positive/negative boundaries of ranges (edge cases) multiple simultaneous boundaries (corner case)
Regression Testing - new code does not break old code

1.4 C++

Bjorne Stroustrup -invented the idea of a class and created ”C with classes”
How can we tell an attempted read failed? If a read fails then cin.fail() is true if a read fails because EOF then cin.fail() is true and cin.eof() is true.
Example: Read all integers from stdin and echo to stdout Stop if non-integer or EOF is encountered.
There is an implicit conversion from cin (iostream) to void * Since a pointer is some numeric value, it can be used in a condition Thus we can use just cin inside a condition cin is true if !cin.fail()
We are using ¿¿ as the ”get from” (input) operator Which also does bit shifts. Also cin ¿¿ a is an expression that returns cin When a read fails, a flag goes up in cin - the flag stays raised until we acknowledge a failure cin.clear() - lowers the flag cin.ignore() - skips over to the next character
C++ provides the type std::string for helping with string stuff (such as reading in strings)
this will read in characters until it hits a whitespace, then its done. i.e. go past any whitespaces

until it hits the first whitespace Reads the next characters into a String, stopping at the next whitespace, ignoring any leading whitespace. Aside: to read the entire line (including whitespace) getline(cin, s) cin for strings is similar to scanf with %s format specifiers in C

Notice that cin ¿¿ i reads in an integer cin ¿¿ s reads in a string The exact same code to read either in, the only difference is the declared type (e.g. type of i or s variable)

C-style format specifiers do have their benefits int i = 95; cout ¡¡ i ¡¡ endl; // just prints out 95 in decimal What if we want hexidecimal, binary, boolean (true/false literal)? We use I/O manipulators to do that. Examples

cout ¡¡ hex ¡¡ i ¡¡ endl; (note hex is std::hex, we are working with namespace std) Sending hex to cout does not print anything. It makes cout change in way that says print any integers that are sent in as hexidecimal form Unless we do cout ¡¡ dec, to change the behaviour back to decimals

Moving along, what if we don't want to read in from keyboard (stdin). This "stream" abstraction we've been using as cin (specifically istream and ostream - in and out respectively) we can use the same way to open and "stream" a file. To read or write from files, we have file streams. They behave just like input and output streams.

The only difference in this example is how we initialize the file stream, requiring a file name! ifstream f("suite.tx"); //variable declaration and initalization type: ifstream varaible name: f parameters for initializing: "suite.txt" (string of filename)

We can attach a stream to a string and read/write from it include ¡sstream¿ So we have an istringstream -read from a string And ostringstream - write to a string

int low... int high... ostringstream ss; ss ¡¡ "Enter an integer between ¡¡ lo ¡¡ " and " ¡¡ high;

Default Arguments: in c++ we can specify default arguments void printSuiteFile(string filename="suite.txt) ¡- is the default value

string s = ss.str(); cout ¡¡ s ¡¡ endl;

Benefit: Convert a number to a string How about reading a string? We can use an ostringstream

Strings in C++ In C we didn't have a built in string type, so instead we used arrays of characters. - needed careful memory management - was very easy to overwrite the null terminator

In C++ we do have a string type - not built in, but part of standard library std::string type -manage their own memory - safer to use string s = "sdgd"; can assign c string to a c++ string and c++ does a conversion

equality: c++ has all of the comparison type operators (==, ¿= ¡=)

ability to refer to individual characters In c++ we can still do s[0], s[1] for characters of a string

string concating in C: strcat in c++ s1 + s2 works!

Note: to get a c style string from a c++ string, we need to use string.c_str()

Overloading: Example: write functions that negate integers and one that negates bools

Pointers int n = 5; int *p = &n; cout ¡¡ p ¡¡ endl; //prints the pointer to n cout ¡¡ *p ¡¡ endl; // prints the int 5

int **pp; pp is a pointer to a pointer to an int. pp = &p;

Arrays Not to be confused with pointers int a[] = 1, 2, 3, 4 this way has a few restrictions of whats needed. the array needs a contiguous block of memory The name of the array is short form for the address of the first element in the array. a = &a[0] *a = a[0] -¿ first element

Structs

C syntax Struct Node int data; Struct Node * next; ;

Constants const int maxGrade = 100; Make as many tihngs constance as possible A constant must always be initialized

const int * p = &n; p is a pointer to a constant integer what you cant do is change the value of n through p

int * const p = &n p is a constant pointer to an integer
cin behaves similarly to scanf for scanf we pass the address of the variable we want to read into although cin ¿¿ x takes in not a pointer, it still acts as if it was given one
CPP has another pointer like type - a refernce int y = 10; int &z = y; z is a reference to y z is a constant pointer to y
a reference is a constant pointer with automatic dereferencing z = 12; (NOT *z = 12 - but this 'does' the same thing)
int *p = &z What happens here even is that p gets the address of y. In no case does z have an identity of it's own.
Cannot have a reference uninitialized its a constant pointer -¿ constant -¿ must be initialized
Can only create a reference to something that has an address. Note: anything that has an address is called an lvalue
Cannot create a reference to a pointer Cannot create a reference to a reference Cannot create a reference to an array Cannot create a reference to a pointer
Pass by Value struct ReallyBig f(ReallyBigStruct) results in copying over the entire struct In C we could pass a pointer to suppress the entire copy In CPP we can also use a reference to do similarly. void h (const ReallyBig &rb) will ensure rb does not get changed outside of scope.
Dynamic Memory Allocation CPP provides new and delete for allocation and deallocating. New is type aware, so it's less prone to errors
Node *np = new Node; as simple as that Since it is type aware it knows how much memory is needed for Node Once done, to reclaim memory delete np;
Dynamically allocating an array cin ¿¿ n; int * arr = new int[n], number of elements in the array Once done, to deallocate the array delete [] arr;
Operator overloading Give a special meaning to C++ operators for types we construct

Preprocessor - a program intercepts the code before it is compiled - transforms code

#include ¡iostream¿ this is a preprocessor directive This tells the preprocessor to paste the contents of iostream right here Look for iostream in the standard place that contains the cpp library

Also #include “...” - look for this file in my current directory

Another directive #define VAR VALUE tells the preprocesor to setup a variable called VAR with the value VALUE

The define directive orders the preprocessor to replace any occurrence of VAR with VALUE -does a search and replace

#define MAX 10 int array[MAX]; =¿ int array[10];

Define is a mechanism to specify constants. It’s completely useless now as C and CPP now have const variable types which made define obsolete.

Preprocessor does nto know (or care) what would get generator by doing the ”FIND AND REPLACE” of define.

Define constants are useful for conditional compilation Examples: In unix it might be int main()

While windows might require code to be int WinMain()

So instead you can use preprocessing to just change the code immediately, as so

#define Unix 1
#define Windows 2
#define OS Unix
#if OS == Unix
int main
#elif OS == Windows
int WinMain()
#endif

#define VAR -define a variable VAR - it’s value is the empty string
#if def VAR
blagh
#ifndef VAR
blagh
#endif

Note doing #if 0 sadasfaf afasf asfasf #endif -super duper way of commenting out code

Moreso we can specify defines from command line. e.g. g++ program.cc -DOS=Unix -o program

Will #define OS Unix for you through the command line

Mind you if you do #define DEBUG -¿ gets the empty string value g++ -DDEBUG *.cc -¿ value is given 1

Separate compilation -split programs into composable modules Interface: .h type definitions, fucntion headers... Implementation: .cc, actual implementation

Ways to compile g++ *.cc (glob all files), if all .cc files are part of the program and you dont compile - header files included inside .cc - never include cc file

g++ argument ”-c” just compile. creates an object file, binaries for code ld (linker) is the matchmaker that links the files together

Hard to keep track of whats being included have IFDEFS in our headers!

Classes Big innovation of OOP You can put functions inside a struct A class is a struct type that can have functions For now we will only regard classes as a struct which contains functions

Functions defined inside a class are called ”member functions” or methods

Say we have a struct Student with a function grade(); When we access the structs fields inside it’s

methods, we may use "final" "midterm" as variables, Where a when we access the method from the struct we are specifically detailing that structs instances
The distinction between fucntions and methods is that a method contains a hidden (implicit) parameter called "this" which is a pointer to which the object that this pointer was called
Initializing Structs Student billy = 60, 70, 80 C style initialization of an object - restrictive
CPP provides th abiity to write constructors which are functions used to construct objects
Struct Studenet int assigs, mid, final;
note fcn name is same as class name Student (int assigs, int mid, int final) this.assigs = assigs; this.mid = mid; this.final = final;
Now using our constructor Student billy(60,70,80); Student billy = Student(60,70,80);
And on the heap Student * pBilly = new Student(60,70,80);
Advantages of writing constructors - full CPP functionality is available -default parameter values
Student(int assigs = 0, int mid=0, int final=0) regular constructor
If you want to default all parameters (so pass nothing) You want to do Person per; and it uses all of the default parameters for the constructor
Initializing Objects every class comes with a default no argument constructor. - calls default constructors on fields (if they have constructors)
e.g. Vec v; // default constructor ran
Note: the builtin (default) constructor goes away as soon as you write any constructor for the class
Once we've defined a struct with our own constructor, we can no longer do
Struct Vec int x; int y;
Vec (int xx, int yy = 2) this-¿x = xx; this-¿y = yy; ;
Vec v1(1,2); Vec v2 = new Vec(1,2); Vec v(1);
Vec v; // won't compile Vec v (1,2); //ok! Another thing you loose when you build your own constructor is c-style initializing Vec v = 1,2 ; // wont compile
Q: What if my class has constant as reference fields? struct MyStruct const int myConst; int &myref; ;
COnstants and references must be initalized! Ok, to initalize them
struct MyStruct const int myConst = 5; int &myRef = z; ;
inializers are not allowed though for fields! (won't compile) What you most likely want is that each object gets it's own constant independant of the class Each instance of myStruct gets its own myConst and myRef Why should they have the same value?
Q: Can we initliaze these inside the construcotr body? A: No, it's too late. by the time the constructor body executes constants and references should have already been initialized
Struct Person
int x; int y; Struct Wallet w;
;
new Person(); Wallet w = new Wallet();
When an object is created, 1) space is allocated (stack or heap) 2) fields are initialized: default constructors are executed for all fields that have default constructors 3) constructor body runs
hijack step 2: done by a member initialization list
Struct MyStruct const int myConst; int &myref;
int x;
MyStruct(int c, int &r, int xx): myConst(c), myref(r), x(xx) CONSTRUCTOR ; -This sort of syntax is only allowed for constructors -You are not restricted to just constants and references
Another example of the member list Struct Student int assigs, mid, final; Student(int assigs, int mid, int final): assigs(assigs), mid(mid), final(final) ;

In this case note that in field(param), FIELD is ALWAYS just looked at as a field, and PARAM is ONLY looked at as parameters in this syntax Another advantage of member initialization lists is that they [can] be more efficient then initializing inside the constructor body (due to default initilization being run and initializing all fields twice if re initializing in cosntructor body) [if all the fields are primative types they won't be initialized and it won't make a difference anyways as primate initiliazation isn't much]
Fields are initialized in the order in which they are declared -irrespective of the order appear in the in the members initialization list -most compilers give you a warning if your order in the initialization list was different than the order of declaration
more initialization stuff Student billy(60,70,80); and we have bobby is a copy of billy Student bobby = billy; To construct a new object as a copy of an existing object we use the copy constructor
-you get a default copy ctor for free (makes a copy field to field) Every class comes with: -default constructor -default copy constructor - copy assignment operator - destructor
Struct Student int assigs, mid, final; Student(int assigs, int mid, int final): assigs(assigs), mid(mid), final(final)
Student(const Student &other): assigs(other.assigs), mid(other.mid), final(other.final) ;
Student bob(1,2,3); Student billy(bob);
Default copy constructor might not do what you expect it to do
Struct Node int data; Node * next;
Node (int data, Node *next): data(data), next(next)
Node (const Node &other) : data(other.data), next(other.next) ;
Node *n = new Node(1, new Node (2, new Node(3, 0)));
Node m = *n; // runs copy Node *p = new Node (*n);
I wanted three linked lists that are copies of eachother Unfortunately we only got NEW first nodes, but the rest of the list is the same for each list (points to the same elements) We get whats called a shallow copy
DEEP COPY Struct Node
Node(const Node &others): data(others.data), next(other.next ? new Node(other.next) : 0) ;
When copies are made 1) Copy constructor: create an object from an existing one default copy constructor copies all fields (shallow) Sometimes we need deep constructors Example of copy: student bobby = billy; // copy constructor
2) when a function passes an object by value 3) when a function returns an object The copy constructor must take others by reference - infinite recursion if called by value (attempts the copy constructor as it passes in)
Destructor When an object is destroyed a method called the destructor is run - a stack allocated object is destroyed when it goes out of scope - a heap allocated object is destroyed when it goes out of scope - default destructor is built in. calls the default constructor on fields if available
Node * np = new Node(1, new Node(2, new Node(3,))); 1) -np will go out of scope - no destructor got called because np is a pointer - linked list still there, so we have a memory leak 2) delete np - calls the default constructor for the object *np, (np points to) - node 1 is deleted, node 2 and 3 are leaked
To reclaim all memory, we must write our own destructor struct Node int data; Node * next;
Node() //calls destructor on all fields delete next;

1.5 Seperate Compilation for Classes

interfaces (.h file): struct definition and function headers implementation (.cc file): full implementation

Node.h #ifndef NODE_H #define NODE_H stuct Node int data; Node * next; Node (int data, Node * next); Node(const Node &other); ; #endif

Node.cc #include "Node.h" Node(int data Node *next): data(data), next(next) Node(const Node &other): data(other.data), next(other.next ? new Node(*other.next) : 0)

-But here we have that the compiler is unable to determine that these are methods defined in Node.h

-Prefix the method with Node::!

Node.cc #include "Node.h" Node::Node(int data Node *next): data(data), next(next) Node::Node(const Node &other): data(other.data), next(other.next ? new Node(*other.next) : 0)

:: is known as the scope resolution operator Another option is using namespaces to do the same thing for the entire .cc context

1.6 Assignment operator

Student billy(60, 70, 80); //constructor with 3 args Student bobby = billy; // copy constructor Student jane; // 0 argument constructor jane = billy; // jane already existed What happens is jane is modifying it's fields to be come a copy of billy (not using copy constructor, no new object is created)

You get a default assignment operator! Need to write your own if dealing with dynamic memory

```
struct Node {
        Node &operator=(const Node &other) {
                data = other.data;
                next = other.next;
                return *this;
        }
};
```

Want a deep copy

```
Node &Node::operator=(const Node &other) {
        data = others.data;
        *next = (others.next) ? *others.next : 0; % mine
        next = new Node (*(other.next)); % on board
        next = others.next ? new Node(*others.next) : 0; % board, null fix
        return *this;
}
```

Memory leak: not deallocating whatever next originally points to

```
Node &Node::operator=(const Node &other) {
        data = other.data;
        delete next;
        next = others.next ? new Node(*others.next) : 0;
}
```

When implementing the operator= always check for self assignment if (this == &others), meaning if they are the pointing to the same address and same object return *this;

If new fails (no memory), nex is a dangling pointer. what we want is ALL OR NOTHING
Delay the delete

```
Node &Node::operator=(const Node &other) {
        if (this == &other) return *this;
        Node *tmp = next;
        next = other.next ? new Node(*other.next) : 0;
        data = other.data;
        delete tmp;
        return *this;
}
```

LECTURE JUNE 19
Assignment operator continued
Copy and swap idiom for assignment operator

```
struct Node {
        void swap(Node &other) {
                int tdata = data;
                data = other.data;
                other.data = tdata;
                Node *tnext = next;
                next = other.next;
                other.next = tnext;
        }
        Node &operator= (Node &other) {
                Node tmp = other; // copy ctor, deep copy
                swap(tmp);
                return *this;
        }
};
```

Is there a memory leak? No, because tmp is stack allocated and the destructor is called when tmp goes out of scope (assuming a properly implemented destructor)

Rule of 3
If you need to write a custom version of

- copy constructor
- destructor
- operator=

then you usually need to write all three
operator= is a member function When an operator is declared as a member function, *this* plays the role of the left hand operand

```
struct Vec {
        int x,y;
        Vec operator+(const Vec &other) {
                Vec v(x + other.x, y + other.y);
                return v;
```

```
        }
        Vec operator*(const int k) {
                Vec v(x * k, y * k);
                return v;
        }
};
```

If we want to do scalar multiplication with the reverse we have to have a standalone function outside of the class

```
Vec operator+(const int k, const Vec &v) {
        return v * k;
}
```

CPP tells you which operators MUST be implemented as a member function

- operator=
- operator[]
- operator-¿
- operator()
- operator T(), where T is some type

```
struct Vec{
        ostream & operator<<(ostream &out) {
                out << x << " " << y << endl;
        }
};
```

Note the vector has to be on the left hand side as this is a member function (which can be annoying) Arrays of objects

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

Vec vectors[3];
Vec * myvectors = new Vec[10];
```

But both of these methods don't compile as they need the default constructor for the vectors to initialize them all.
Stack allocated array: - use aray initialization

```
Vec vectors[3] = { Vec(1,2), Vec(3,4), Vec(5,6) };
```

Heap allocated array: - give a default constructor - create an array of pointers to objects

1.7 Constants revisted

```
void f(const Node &n) { ... }
```

- this means you cannot modify the values of n's fields Q: Can I call methods on a const object A: yes you can, as long as the method promises it's behaviour (takes in const also)

```
struct Student {
        int assigs, mid, field;
        float grade() {
                return assigs * 0.4 + mid * 0.2 + final * 0.4;
        }
        // const objects can call const methods, we have to explicitly ↩
            state const3
};
```

Want to track how many times grade is called on each student

```
struct Student {
        int assigs, mid, final;
        int numMethodCalls;
        float grade const() {
                ++numMethodCalls;
        return ...;
        }
        // want to be able to modify numMethodCalls by delcaring the ↩
            field mutable
        // mutable int numMethodCalls;
};
```

1.8 SE Topic: Design Patterns

If you have a situation like THIS, then THIS is a good programming technique to solve the problem. Singleton Pattern We have a class C, and we want to ensure only one instance of C is ever created, no matter how many times we request a new instance C++ static members Static field: is associated with the class itself and no instance of the class - per class rather than per object

```
struct Student {
        static int numInstances;
        Student(...) ... {
                ++numInstances;
        }
};
```

This will keep track of how many students there are
Initializing the static variable in the .cc file

```
int Student::numInstances = 0;
```

Static member functions (or static methods) - not dependant on any object (no this parameter) Restrictions on static member functions - can only access static fields - can only call other static methods

```
struct Student {
        static int numInstances;
        static void printInstances() {
                cout << numInstances << endl;
        }
};
```

It can reference numInstances because numInstances hasalso been declared static

```
Student Billy(...);
Student Bobby(...);
Student::printInstances();
```