**CS246 Midterm**

**Linux shell:**

**- Shell**

Interface of operating system

- Open files, more files

**- Linux File System**

- Contains files

- Some files can obtain other files

- Directories:

The file system forms a tree, rooted at a directory called “root”

**- Linux Commands Handout e.g. cd, pwd, ls, echo, rm,**

-Linux command summary.pdf

**- Globbing patterns**

- \* match anything e.g. \*.txt matching all the txt files

- ? match one single character

- [abc] match one character from the

- [a-z] match one character from the range

- [!a-z] does not match any character

**- Cat**

- displays the content of a file by calling cat <filename>

**- Input/Output Redirection**

- Input Redirection:

Direct input into a program from a file using <

e.g. ./program arg1 arg2 < test.in

- Output Redirection:

redirect the output of a program or command to a file using >

using >> to append the output to the file

e.g. ./program arg1 arg2 > test.out

**- Pipes**

- Connects the stdout of one process to stdin of another process using |

e.g. cat words\*.txt |sort|uniq

**- egrep**

- extended global regular expression print, usually used for

searching within a file by calling “egrep <pattern> <files>”

- The <pattern>s are “regular expressions”

- Output of egrep: lines from the files that match the pattern

- some terms in regular expression:

- | means the previous or the other one will be valid

- […] chose one from the set

- () specifies a specific range e.g. (c|C)s246 --- cs246 or Cs246

- ? 0or1previous subexpression e.g. “cs?246” --- 246 or cs246

- + 1 or more of previous subexpr

- \* 0 or more of previous subexpr e.g. cs\*246 --- 246, cs246,

cscs246, cscscs246…

- ^ choose anything but the one e.g. [^a] --anything but a

**- File Permissions (chmod)**

- using command “chmod” to make a file accessable

- ownership class: u—users, g—groups, o—others, a—all

- operator: + :add, -: remove, =: set to

- usually using “chmod a+x <filename>” to make a file executable,

using “chmod a+r” to make a file readable, using “chmod a+w” to

make a file writable

**- Shell Variables ($PATH)**

- to define a shell variable: $>x=1 with no space between

- variable could be a string value

- double quotes allows use of variables, single quotes does not

- using $before the variable name e.g. echo $x and echo ${x} are

same

- there is some default global constants, namely $PATH

**- Shell Scripts**

Text file containing linux commands/programs that is executable as

a program.

- a shebang line “#!/bin/bash"

- Command Line Arguments to a script is -- $0: the name of the

shell, $1: first variable, $2: second variable …

e.g. echo “$1” $(ls) – prints all the names in the current directory

e.g.

usage: ./goodPassword password

#!/bin/bash

egrep “^$1$” /usr/share/dict/word > /dev/null

**- $#, $?, /dev/null, $0**

- $# represents the number of arguments in the shell script, $# and

${#} are the same

- $? Represents the status code, $? Is 0 if the previous command is

true; is 1 if the previous command is false

- /dev/null to discard output into null

- $0 – contents before any argument is typed(usually the script

name)

**- script functions**

-usually follows the format “ <funcname> (){ <function body> }

e.g.

usage () {

echo “Usage: $0 password” > &2

exit 1;

}

**- if statement**

- using =, != for string comparison

- using –eq, -ne, -le for number comparison

- using –e for exists

- follows the syntax of:

if[ condtion ] ; then

[ cotent]

elif[ ] ; then

[]

else

[]

fi

e.g.

if [ $? –eq 0 ]; then //note the white spaces

echo Not a good password

else

echo Maybe a good password

fi

**- While loop**

- a example:

Print from 1 to $1

#!/bin/bash

x=1

while [ $x –le &1]; do

echo $x

x=$((x+1))

done

**- for loop**

- three examples:

example1:

for i in a b c d; do echo $i; done

this will print a, b, c, d on each line

example 2:

renameC

#!/bin/bash

# Rename all .c file to .cc

for name in \*.c; do

mv ${name} ${name%C}cc

done

example 3:

#!/bin/bash

x=0

for word in $(cat $2) ; do

if [ “$1” = $word ]; then

x = $((x+1))

fi

this will count the words in $1

**C++:**

**- C++ : Hello World**

An code example:

#include <iostream>

using namespace std;

int main (){

cout <<”Hello world” << endl;

return 0;

}

**- Compiling/Executing C++ programs**

- using the following bash commands:

$>g++-5 –std==c++14 hello.cc –o myprogram

to compile

Alternatively,

$>g++14 hello.cc –o myprog

**- Stream Objects (cin, cout, cerr)**

- Stream Objects are in <iostream>, when you include <iostream>,

you are importing std::cin(type istream), std::cout(type ostream),

std::cerr(type ostream)

- cin reads from stdin, cout writes from stdout, cerr writes to stderr

**- I/O Operators (>>, <<)**

- C++ provide 2 operators, input operator >> and output operator

<<

e.g.

Add two numbers intro/plus cc

#include <iostream>

using namespace std,

int main () {

int x, y;

cin >> x >> y;

cout << x+y <<endl;

}

**- cin.fail(), cin.eof()**

- cin will ignore white spaces, it reads from first non-white until a

white space

- if a read fails, the expression cin.fail() is true

- if a read fails due to EOF, the expressions cin.fail() and cin.eof()

are both true

e.g. Read all ints from stdin echo one per line to stdout, stop if read

fails

int main(){

Int I;

While(true){

Cin >> I;

If(cin.fail()){

Break;

}

cout << I <<endl

}

}

**- implicit conversion of a stream to bool**

- C++ defines an automatic conversion from cin to bool

- cin is true if cin.fail() is false

e.g.

int main(){

I nt I;

while(true){

cin >> I;

if(!cin) break;

cout << I <<endl;

}

}

**- << and >> are binary operators: must produce an expression**

e.g. “cin>>I: is binary expression, it produces cin

**- << and >> cascading**

e.g. a cascading of cin:

cin>>x>>y; 🡺 cin>>y; 🡺 cin;

- if a read fail, all subsequent reads fail

**- cin.ignore(), cin.clear()**

- if a read fails due to a non-int, it fails forever and the flag is still

up

- To lowers the fail flag we need to use cin.clear()

- In most cases use only cin.clear() is not good enough since the

offending input is still waiting to be read

- We use cin.ignore() to ignore one character

e.g. Read all ints & echo to the stdout, skip non-int input

int main(){

int I;

while(true){

if(cin>>i){

cout << I <<endl;

}

else { // read failed

if(cin.eof()) break;

cin.clear();

cin.ignore();

}

}

**- std::string**

- C++ has a string type, to use string we need “#include <string>”

**- Semantics of reading from cin**

- when reading a string, cin will read until it has a white space

- To read strings that include spaces using getline(cin, s), it will read

from current characters until a newline

- cin can be used to read in int/string etc without needing to

remember format specifies

**- I/O Manipulators: e.g. hex, dec, showpoint, setprecision , boolalpha, header <iomanip>**

- In C++, we use IO manipulator to specify the type of output

- some manipulators:

- hex, cout<<hex makes the output in the form of hex

- dec, cout<<dec makes the output to decimal

- setprecision, used for floating-point on output operators

- showpoint, shows the decimal point

- boolalpha, When the boolalpha format flag is set, bool values

are inserted/extracted by their textual representation: either

true or false, instead of integral values

**- Stream Abstraction for files: <fstream>, ifstream, ofstream**

- The “stream abstraction” can be used on Read/Write to files, to

use it we need to include “<fstream>”

- ifstream: read from files, ofstream: write to files

usage example:

e.g. Read of output contents of a file io/readfile.cc

int main(){

ifstream(type) filestream(var name){“myfile.txt”}(initial value);

while(filestream >> s){

cout << s <<endl;

}

}

- the file is opened as part of initialization, it is automatically closed when

filestream variable goes out of scope

- you can use ifstream variable exactly like cin, use ofstream variable exactly

like cout

**- Stream abstraction for strings: <sstream>, istringstream, ostringstream**

- String can be used as a source of data

- by including <sstream>, we can use istringstream for reading

from string, ostringstream for writing to a stream

ostringstream e.g.:

Ostringstream os;

//Os<”enters a value btw”>

Os <<hi <<”and” <<lo <<endl

string s = os.str();

**- Converting a string to an integer**

- a string can be converted into an integer through sstream

e.g.

int main () {

int num;

while(true){

cout << “enter a number” << endl

string s;

cin >> s;

istringstream ss{s};

if(ss >>num) break;

cout<< “Not a number” << endl;

}

**- Strings in C++**

- C++ provides string type, it automatically resize if needed

- we can use ==, !=, <, >, <=, >= for string coparison

- we can treat string as array as well, e.g. s[0], s[1]…

- we we can also use + to append strings

**- string operations: concat, length, comparisons, length**

- as described above

**- Default Arguments**

- we can make default arguments by assigning parameters default

values in the function signatures,

e.g. void test (int name = 0, string str = “bla”)

- some restrictions:

- cannot make a parameter with no default value follow a

parameter that has default value

e.g.

~~void test (int num = 0, string str);~~

- If you are going to leave out argument, it must be the last

e.g.

void test(int name = 0, string str = “bla”)

test();--valid

test(1);---valid

test(1, “bla”); ---valid

test(“bla”);----invalid

test(, “bla”); ---invalid

**- Function Overloading**

- In C++, we can have functions that have the same name as long

as they differ in the number as types of parameters

- we can also overload operators

**- Lvalue References**

- Anything that has a reference is considered a Lvalue

- Lvalue is a storage location(something with an adress), anything

that can appear on the left hand side pf an assignment operator

can be a Lvalue

- anything that can only appear on the RHS of = is considered a

Rvalue

**- Pass by reference**

- C++ has a “pointer-like” type: references

- References are like constant pointers with automatic

dereferencing

- You cannot leave reference uninitialized, a reference

must be initialized e.g. int &z;--invalid

- You cannot create a pointer to a reference, but you can create

a reference to a pointer

- You cannot create an array of references

e.g.

void f (int &n){ … }

void g (cons int &n){ … } // allows to be called by rvalue

~~f(5)~~

~~f(x+y)~~

g(5);

g(x+y);

**- Pass by Value vs Passing a pointer vs Pass by reference: pros and cons of each**

- If we want to avoid copy, the only option in C would be to pass

the address, in C++ we could add the option to pass by reference

- By passing by references or pointers we are also able to modify

the value itself

- Passing by Reference to const is a good style since it prevents the

copy(efficient) and diallows updates to the value

**- Dynamic Memory Allocation: new and delete**

- In C++, we use “new” and “delete” key words, “new” for memory

allocations and delete to free the memory

- Heap memory lives on after a function returns, forgetting to free

or delete allocated memory leads to “memory leak”

- we call “delete[]” when deleting an array

**- Operator Overloading, Examples: Vec, Grade**

- we can overload operators

- example of Vec:

Vec Operator + (const vec &lhs, const vec &rhs){

Vec v3 = {lhs.x +rhs.x , lhs.y+rhs.y};

Return v3;

}

Vec operator \*(int k, const vec &v){

Return {k\*v.x, k\*v.y};

}

Vec operator \*(const vec &v, ink k){

return k\*v;

}

- example of Grade:

struct Grade{

int mark;

};

ostream &operator <<(ostream out, const Grade &g){

out << “You grade: “ << g.mark << “”; (no new line)

return out;

}

Istream &operator >>(istream &in, Grade &g){

In >> g.mark;

If(g.mark <0) g.mark = 0;

If(g.mark >100)g.mark = 100;

Return in;

}

**- The C and C++ Preprocessor (#include, #define, Using #define for Conditional Compilation)**

- Before the compiler sees the code, the preprocessor runs first

- # --- preprocessor directive

- #define VAR VALUE: search and replace all occurrences of VAR

with VALUE(discouraged, using const instead)

- #ifdef VAR : define if VAR is defined

- #ifndef VAR: define if VAR is undefined

e.g.

#ifndef VECTOR\_H

#define VECTOR\_H

(old vector.h)

#endif

**- Separate Compilation**

- By default g++ tries to compile, link to produce executable file

- we use “g++ -c <filename>.cc” to produce a <filename>.o, use

“g++ <filename1>.o <filename2>.o <executable>” to produce

a executable file

**- C++ Classes**

- Classis an important part for OOP programming, basically classes

are structures containing functions

- An instance of a class is called an object

- You can only call a method of a class using an object of the class

- A method has a hidden parameter called “this”, “this” is a pointer

to the object on which the method was called

**- Initializing Objects**

- C style initialization values must be constants

e.g. Student billy{60, 70, 80};

- In C++ we can write methods called constructors to construct

objects

**- Constructors**

- Advantages of constructors:

- they are functions(if, for etc)

- default values to parameters

- sanity check and class invariant

an example of a constructor:

Struct Students{

Int assns, mt, final;

Float grade(){…}

Student (int assns.; int mt, int final){ (No return type) (name of class)

This->assns = assns.;

This->mt = mt;

This->final = final;

}

};

To call the constructor:

student billy{60, 70} –final 0;

student billy{} // all 0

student billy; // all 0

**- Default ctor**

- every class comes with a default (no parameter) constructor

- the constructor calls default constructors on fields that are

objects

**- Initializing const and reference fields**

- Steps for object creation:

- space is allocated on stack/heap

- Fields that are objects are default constructed

- Constructor body runs

- We have to initialize constant values and reference fields inside

the structure definition since fields must be constructed before

constructor body runs

**- Member Initialization List (MIL)**

- However, by changing step 2, we are able to define constants and

references through constructor calls

- Member Initialization List(MIL) allows us to do so, it is not

restricted to consts and references.

- In MIL, the thing outside each pair must be a field

- Fields in MIL are always initialized in class declaration order

- Using MIL can be more efficient that using constructor body

(default constructors will not run if a MIL happens)

e.g.

Struct Student{

Const int id;

Int assns mt, final;

Student(int id, int assns, mt, int, final)

: id{id}, assns.{assns}, mt{mt}, final{final}{}

};

**- Copy Ctor**

- Copy Constructors are used to construct an object as a copy of

existing object

- There is a default copy constructor in every class

- A copy constructor always takes a single reference of the class as

a parameter

e.g.

Struct Student{

Int assns., mt, final;

Student(const Student &other)

: assns{other.assms}, mt{other.mt}, final{other.final}{}

}; // free copy ctor

- However, sometimes we need to deep copy some fields in a class for

some structures, namely Node(otherwise constructors would only

make a shallow copy)

e.g.

struct Node {

Node(const Node &other)

: data{other.data}, next{other.next? new Node {\*other.next} : nullptr} {}

}; //deep copy

- copy constructors are called when:

- constructing an object as a copy of another

- a function has a parameter that is passed by value

- when something is returned by value

- Parameters of a copy constructor must be a reference, cannot be a

value

**- explicit keyword**

- The compiler is allowed to make one implicit conversion to resolve

the parameters to a function. What this means is that the

compiler can use constructors callable with a single parameter to

convert from one type to another in order to get the right type for

a parameter. By using “explicit” keyword, the compiler is not

allowed to do so.

e.g.

struct Node{

int data;

Node \*next;

Node(int data) :data{data}, next{nullptr}{}

};

Node n{4};

Node n =4;//legal

Void foo(Node n){…}

Foo(4); //legal

Explicit (int data) :data{data}, next{nullptr}{}, then Node = 4 and foo(4) are

illegal

**- Destructors (dtor)**

- When an object is destroyed, the destructor runs when:

- a stack allocated object goes out of scope

- heap allocated object is deleted

- 3 steps to destroy an object

- Destructor body runs

- fields that are objects are destroyed

- space is deallocated

- A class has only one destructor, named “~<Classname>()”

- Every class comes with a destructor, it simply called destructors

on field that are objects

e.g.

Struct Node {

….

~Node(){

delete next;

}

};

**- Copy Assignment operator ( operator=)**

- We need to overload = operator if we want to assign values of an

object to another one, this is called Copy Assignment Operator

- Since = updates an existing object, next could already be pointing

to heap allocated memory, we must delete it first

- using swap can prevent deleting memory

e.g.

#include <utility>

Struct Node {

…

void swap(Node &other){

using std::swap;

swap(data, other.data);

swap(next, other.next);

}

Node &operator= (const Node &other){

Node tmp = other; // relies on deep copy ctor

Swap(tmp);

Return \*this;

}

}

**- Rvalue References**

- Rvalues are usually temporary values that are going to be

destroyed

- Rvalue reference: reference to a temporary

e.g.

Node n1 = plusOne(n);

Actual process of compiling:

Tmp <- plusOne(n);

Node n1 = tmp;

**- Move Ctor**

- Move Ctors usually contains only one parameter, which is a

rvalue reference(using &&)

e.g.

Node (Node &&other)

: data{other.data}, next{other.next}{

other.next = nullptr;

rvalue reference --- a temporary

-it’s about to be destroyed

-it’s dtors will run and deallocate the linked list

}

**- Move Assignment Operator**

- When you need something to be assigned by a rvalue

e.g.

Node &operator = (Node &&other){

Swap(other);

return \*this;

}

**- Superficial knowledge of Copy/Move elision**

- A compiler is allowed to skip copy/move ctor in certain situations

even if the ctor have side effects

- To turn off the optimization:

g++14 –fno-slide-constructor (files)

**- Rule of 5**

- If you write a custom version of:

-Copy ctor

-Copy assignment operator

-Dtor

-Move ctor

-Move assignment operator

Then you typically need to implement all of them

**- Member functions vs Standalone functions**

- operator = must be implemented as methods

- LHS is the “this” pointer

- we can also implement other operators as methods

e.g

Vec operator\* (int k){

return {x\*k, y\*k};

}

- For I/O operators, we should not make them as methods since

it makes writing outputs confusing as we need to filp the LHS

and RHS

- operators must be implemented as methods: =, [], (), ->

**- Arrays of Objects**

- To create an array of objects we could:

- provide default constructor

- using stack array initialization

- stack / heap allocation (Array of pointers to objects)

**- Separate compilation of classes (:: the scope resolution operator)**

- Methods headers go into the header files, methods

implementation goes into .cc files

- :: is the scope resolution operator, e.g. int Node:: getdata(){};

**- const Methods**

- If you declare a method to be const, you are promising not to

change the fields of the object on which the method is called

- If you make a const object, make sure your methods are constant

before calling them

e.g.

const strudent billy{60, 70 ,80};

cout << billy.grade() << endl;

// will not compile

billy is const

grade does not promise it will not change fields

**- mutable keyword**

- If we add “mutable” keyword to a field, we basically make the field

mutable, whereupon const methods can modify the mutable field

e.g.

Struct student{

Int assns., mt, final;

Mutable Int counter = 0;

Float grade() const

{ ++counter;

return …..;

}

}

**- static keyword in C++**

- A static field is associated with the class, not the object of the

class(similarly to global constant)

- static fields must be defined in an extern file

- static member functions can only access static fields and member

functions, static functions does not have “this” parameter

**- Invariants**

-A statement or assumption that is supposed to hold true for a class

for it to function correctly

- It is impossible to grantee invariants if a client has full access to

members of class

**- Encapsulation: class keyword**

- Encapsulation:

- objects should act like “black box” – a capsule

- hide implementation details

- clients interact through provided interface

- Keywords for encapsulation: private / public:

e.g.

struct Vec{

Vec(int x, int y): ………..;

//default visibility in a struct is public

private: //change visibility to private

int x;

int y;

public:

Vec operator+()……..

};

int main(){

Vec v{3,4}; //2 parameter constructor is public

Vec v1 = v + v;

Int num = v.x; --- invalid

Int num = v.y;------invalid due to “private”

}

- ADVICE: at a minimum keep all fields private

**- Invariant example: List and Node class**

- List class:

List.h

Class List{

Struct Node;//declared a private nested Node class

Node \*theList;

Public:

Void addToFront(int n);

Int ith(int i);

~List();

};

List.cc

struct List::Node{

int data;

Node \*next;

Node(int data, Node \*next):……….

~Node(){delete next}

};

void List::addToFront(int n){

thisList = new Node(n, theList);

}

int List::ith(int i){

Node \*cur = theList;

For(int j =0; j < I && curr; ++j , cur = cur->next ;){

}

return curr->data;

}

List: ~List(){

Delete theList;

}