CS 246

Subject

- Linux Shell (2 weeks)
- C++03 (10 weeks)
- Tools
- Software Engineering
-
- Module 1: Linux Shell
- Shell is the interface between the OS and the Users.
- - Gets the OS to do “things” for us.
- 1. Graphical Shell
- - Windows, Apple iOS
- - Linux (Gnome, KDE)
- Adv: intuitive, easy to learn
- Dis: not very powerful
- 2. Command-line Shell (text-based)
- - primary shell for Unix/Linux
- Adv: powerful
- Dis: have to learn those commands
- Brief History:
- The original unix came with a command-line shell: shell (Bourne Shell)
- - C shell (csh)
- - K shell (ksh)
- - Turbo C Shell (tcsh)
- - Bourne again Shell (bash)

- $> echo $0
- bash
- %> bash
- bash
-
- Linux File System
- Tree Structure
- - Files: programs, data
- - Directory: can hold other files
- Root Directory: /
- We can specify any file by giving its PATH from the root directory.
- /bin - bin inside the root directory
- /usr/bin - bin inside the usr directory inside root.
- Current Directory: which is the directory we are currently “sitting” in.
- pwd: present working directory, command to check current directory.
-
- /home/hobo/cs246/fall/a0 is absolute path, starts at root directory
- Relative Path: w.r.t the current directory
- Suppose my pwd i /home/hobo/cs246
- the fall/a0 (relative path) is the same as the path /home/hobo/cs246/fall/a0
- cd change directory
- Special Directories
- 1. . (dot): Current directory
- 2. ..(two dots): parent directory (cd ../.. change to grandparent path)
- 3. ~(tilde) your home directory (shortcut cd)
- 4. ~userid: userid’s home directory

- To see the contents of a directory ls: listing
- only shows non-hidden files
- Hidden files start with a dot(.)
- ls -a show all files including hidden files
-
- Wildcard Matching
- ls can be supplied with a globing pattern
- ls *.txt match everything ending with .txt
- *match anything
- Working with files
- cat -concatenate
-
- Ctrl+C Kill
- Ctrl+D sends EOF
-
- cat > out.txt
- redirected output to file out.txt
- Input redirection cat < sample.txt

Each program is attached to 3 streams

Standard Input (stdin)->Program Process->Standard Error (stderr) (not buffered)

->Standard Output (stdout) (buffered)

When the process, needs input, it looks towards stdin (default keyboard) < (input redirection) changes stdin to a file

Stdout (default screen) > (output redirection) changes to a file

myprog <in.txt >out.txt 2>err.log

Lecture 2 Pipes

eg. number of words in first 20 lines of sample.txt:

head -20 sample.txt | wc -w

want to give the output of a program as an argument to another command/ program

echo Today is `date` and I I am `whoami`

`date` is equivalent to $(date)

Pattern Matching within Text Files

egrep — extended global regular expression pattern

grep -E is equivalent to egrep

usage: egrep pattern file

output: lines that contain a match of the pattern

Examples

egrep cs246 index.shtml

egrep “cs246|CS246” index.shtml

egrep “(c|C)(s|S)246” index.shtml

-i ignore the case

a|b|c|d = [abcd] = [a-d]

[^abcd] any one character not in the set

how to match an optional expression

? — 0 or 1 of the preceding expression “cs ?246”

* — 0 or more of the preceding expression

^ — line should start with this pattern

$ — line ends here (nothing afterwords)

“^cs246.*cs246$”

dot. — match any one symbol’

use \ to escape special characters

+ — 1 or more of the preceding expression

egrep “(.+)” index.shtml

print all words of even length from /usr/share/dic/words/

egrep “^(..)+$” …

print all files in the current directory whose name contains exactly one a

ls l grep “^[^a]*a[^a]*$”

Lecture 3 Permissions

ls -l — long listing

group:

each user can belong to one or more groups

each file belongs to one group

- rwx rwx rwx

type of file - ordinary file

d directory

l symbolic link

three groups: user (owner of the file), group (all user in the group), other users.

r — read permission

w — write permission

x — execute permission

	file	directory
r	see the contents of the file (cat)	read the contests (ls)
w	file contents can be modified	modify contents add/remove file
x	can execute a file	be able to navigate it (cd)

The owner is the only one with the ability to change file permissions.

These is no permission that grants the ability to change file permissions.

chmod mode file

mode: ownership-class (u(user)/g(group)/o(other)/a(all)) operator(+grant/-remove/=exactly) permission(rwx)

e.g give other read permission: chmod o+r file.txt

e.g revoke execute permission from group: chmod g-x file.cc

Lecture 4

Shell variables

$> x=1

$> echo $x

1. To set a variable do not use $
2. To access the value of a variable, do use $ or ${}
3. Values for variables are always strings

Global variables

PATH colon separated list of directories.

Script: a file containing sequence of linux commands, executed as a program

e.g print the date, current user, current directory

#!/bin/bash

date

whoami

pwd

1. need to tell the shell where the script is
2. script must have execute permission depending on who you are.

Command line arguments

$1 first argument

$2 second argument

e.g check whether a word is in the dictionary

./isitaword hello

#!/bin/bash

egrep “^1” /usr/share/dict/words

e.g A good password is not in the dictionary. Is the argument a good password.

Programs return a status code, 0 for success, non-zero for failure

$? status code of the last run command.

#!/bin/bash

egrep “^1” /usr/share/dict/words > /dev/null (black hole)

if [$? -eq 0]; then

echo Not a Good Password

else

echo Maybe a good password

fi

Lecture 5: Scripts, Testing, C++

e.g. Print the numbers from 1 to $1

```
#!/bin/bash
x=1
while [ $x -le $1 ]; do
        echo $x
        x=$((x+1))
done
if [ $# -ne 1]; then
fi
```

For Loops

e.g. rename all .cpp files to .cc

```
#!/bin/bash
for name in *.cpp; do
        mv ${name} ${name%cpp}cc
done
```

e.g. how many times does a word ($1) occurs in the file ($2)

```
#!/bin/bash
x=0
for word in `cat "$2"` ; do
        if [ $word == $1 ]; then
                x=$((x+1))
        fi
done
echo $x
```

e.g. Payday is the last friday of the month. When is this month's payday?

cal | awk ' { print $6 } ' | egrep [0-9] | tail -1

Testing

Due date 1: test suites

- Marmoset will run your test suite on a correct implementation (public)
- Set of buggy programs

Due date 2: Coding

Output Formatting

Numeric Ranges

- check extremes
- edge cases
- corner cases

C++

Bjarne Stroustrup in 80's - AT&T labs (Unix/C - 70's)

Translate his experience with Simula 67 (OO programming language) into a language which was more popular. — C with classes

C++ is too big. C++03

In c, stdio, printf are still available. Don't do it!! Preferred way in C++ <iostream>

compile $> g++ hello.cc (create executable a.out)

$> g++ hello.cc -o hello (create executable hello)

Lecture 6: I/O

cout << stuff < endl;

C++ provides 3 I/O objects: cout prints to stdout

cin reads from stdin

cerr prints to stderr

2 I/O operators: << output operator

>> input operator

Q: How can we tell whether a read failed?

A: If a read fails then cin.fail() is true

If a read fails due to end of input then both cin.fail() and cin.eof() are true.

@>>3 vs cin>>i

cin has side effects

while(cin >> i) cout << i << endl;

e.g. Read all ints and echo them to stdout. Ignore non-integers. Terminate at eof

```
while(true) {
        if(!(cin>>i)) {
                if(cin.eof()) break;
                else {
                        cin.clear(); //lowers the flag
                        cin.ignore(); //ignore the next thing
                }
        }
        else cout << i << endl;
}
```

Lecture 7 Streams

C++ we have a string data type

#include <iostream>

#include <string>

To read an entire line, use getline(cin, s) read from first unread char in the input stream until we hit a newline.

In C, we would have used scant with %s

How to format out in C++?

Use I/O manipulators

int i = 95;

cout << hex << i;

The stream abstraction applies to other sources of data. Perhaps we want to read from a file

#include <fstream>

using namespace std;

int main() {

```
ifstream f (“suite.txt”);
string s;
while(f >> s) {
        cout << s << endl;
}
```

}

Anything you can do with con (istream) you can do with a variable of type ifstream

We can attach a stream to string

istringstream input string stream, very useful for converting a string to an integer

ostringstream output stream

string s;

istringstream ss(s);

ss >> i;

	In C	In C++
equality	strcmp	s1 == s2
inequality	strcmp	s1 != s2
comparisson	strcmp	s1 < s2
length	strlen	s1.length()
concatenation	strcat	s1 + s2

Default arguments

void printSuiteFile (string filename)

most of the time, print “suite.txt”

once in a while, some other name

Lecture 8: Assorted Goodies

Default Arguments

```
void  printSuiteFile(string filename=“suite.txt”) {
        ifstream myfile(name.c_str()); //name must be a C style string
        string s;
        while(myfile >> s) {
                cout << s << endl;
        }
}
```

printSuiteFile();

printSuiteFile(‘mytests.txt”);

1. optional parameters must appear last
2. Not allowed to follow a default argument with a non default argument.

void testParms(int number = 0, string city = “waterloo”)

testParms(5, “Toronto”);

Overloading (function)

int negate(int a) { return -a; }

bool negate(bool a) { return !b; }

in C, won’t compile. Only looks at the function’s name

In C++, the compiler looks at the name, number of arguments and the types of those arguments.

It is not enough for two functions to only differ on the return type

<< is overloaded;

Declaration before use

In C/C++, must declare something before we use it.

Problem with mutual recursion

In forwardBad, there is no declaration of odd before its use in even (complie error)

In forwardGood, we fixed this bool odd(unsigned int n); forward declaration

tell the compiler that a function with this signature exists.

Difference between a declaration and definition

Declaration: assertion that something exists

Definition: implementation.

An entity can be declared as many time as you want.

Pointers

int n =5;

int *p = &n; //p contains the address of n

cout << p; // print out the address

cout << *p; //print the contents of thing p points to

```
int **pp; //pointer to a pointer to an int;
pp = &p;
**pp = 10;
```

Arrays

```
    int a[] = {1,2,4,8};
```

An array is not a pointer!!

The name of an array is shorthand for the address of the first element of the array.

```
a is &a[0];
*a == a[0];
*(a+1)==a[1];
```

struct

```
struct Node {
    int n;
    Node next; // wrong, the compiler dose not know the size of the structure
    Node *next; // omit “struct” in C++ if you want;
}; // don’t forget the ;!!!!!
```

Constants

```
const int maxgrade = 100;
```

A constant definition must be initialized

```
const Node n = {5, 0};
```

I cannot change n

```
int n = 5;
```

```
const int *p = &n;
//p is a ptr to a const int
p=&m; //correct;
*p=7; //wrong
cannot change the thing p points to using p;;

int * const p = &n;
// p is a const ptr to an int;
p=&m;//wrong
*p=7;//correct
const int * const p = &n;
//p is a const ptr to an int which is constant

Parameter passing
void inc(int n) {
        n += 1;
}
int x=5;
inc(x);
cout <<  x << endl; // print 5;
void inc(int *n) {
        *n += 1;
}
int x = 5;
inc(&x);
cout << x << end; // print 6
```

```
void inc(int &n) {
        n += 1;
}
int x = 5;
inc(x);
cout << x << endl; // print 6;
```

Lecture 9

scanf("%d", &i);

cin >> (&i); we don't send an address

C++ provides another ptr-like type Reference

```
int y = 10;
int &z = y; // z is a reference to y
```

A reference is like a constant pointer with automatic dereferencing

```
z = 12; //legal
```

it sets the value of y to 12.

Reference has no identity of its own. z mimics y

1. cannot leave a reference uninitialized;
2. a reference must be initialized to something that has an address

```
        int &y=z+z//wrong
```

3. cannot create a pointer to a reference

```
        int &*x=…; //wrong
```

4. Cannot create a reference to a reference: int &&x=…; //wrong

5. Cannot create an array of references

```
        int &x[3] = {y,y,y};
```

cin >> x;

istream &operator >> (istream &in, int &data) {}

RHS passed by reference so that change to data is really a change to x

struct ReallyBig {};

int f(ReallyBig rb) {}

rb is a copy of x: 1. expensive. 2. changes to rb are not seen in the caller

In C, we would pas a ptr to ReallyBig to avoid both problems

In C++ we can also pass RallyBig by reference int g(ReallyBig &rb) {}

1. no copy
2. changes visible to caller
3. automatic dereferencing

What I want:

1. efficient - no copy
2. prevent writes int h(const ReallyBig &rb)

Advice:

Prefer to pass by const reference for anything that is larger than an int.

Dynamic Memory

In C, int *p = malloc(size * sizeof(type))

C++ does support malloc/free, You will NOT use them

C++ allocate using new, Deallocate using delete

Dynamic Arrays

```
cin >> n;
int *arr = new int[n]; //new knows that you want an array of n int elements
delete [] arr;
```

Memory associated with a prig contains 3 things

1. program
2. heap (downward)
3. stack (upward)

Variables can be on the stack or the heap

Local variables allotted space on the stack, space is automatically reclaimed when vars go out of scope

Lecture 10

```
Node getMeANode() {
        Node n;
        return n;
}
```

inefficient, returns value is copied.

```
Node *getMeANode() {
        Node n;
        return &n;
}
```

Really bad: returning a ptr to a stack allocated variable

```
Node *getMeANode() {
        Node *np = new Node; //allocate in heap
        return np;
}
```

operator overloading

function overloading:

```
        int negate(int);
        bool negate(bool);
```

>>, << operators are overloaded

+ is overloaded

We can give meaning to C++ operators for any new type we create

```
struct vec{
        int x, y;
};
vec operator+(const vec &a, const vec &b);
```

overload << and >> (return the status)

```
struct grade {
        int thegrade;
};
ostream& operator<<(ostream &out, const grade &g) {
        out << g.thegrade << "%";
        return out;
}
istream& operator>>(istream &in, Grade &g) {
```

```
        in >> g.theGrade;
        if(g.theGrade>100) g.theGrade = 100;
        if(g.theGrade<0) g.theGrade = 0;
        return in;
}
```

Preprocessor

1. source code
2. preprocessor
3. compiler

#include — preprocessor directive

#include <iostream> look standard c++ lib /usr/lib/c++

#include “vector.h” look the lib in current directory

#define directive

#define VAR VALUE — preprocessor variable with VALUE (default is empty string)

#define MAX 10

int myArray[MAX]; => int myArray[10]; // replaced by the presence op const;

#define ever ;;

for(ever) {}

Defined constants are useful for conditional compilation

unit — int main() {}

windows — int WinMain() {}

#define UNIT 1

#define WINDOWS 2

```
#if OS == UNIX
        int main() {
#elif OS == WINDOWS
        int WinMain() {
#endif
```

```
g++ -DX=15 define.cc
X will be defined as 15
```

```
#ifdef DEBUG
        …
#end
```

Lecture 11: Separate Compilation/C++ classes

Preprocessor Review

#include

#define VAR VALUE —conditional compilation

#define VAR

#ifdef VAR

#ifndef VAR

command line preprocessor arguments

        g++ -DX=15 …

preprocessor/define.cc

preprocessor/debug.cc

```
#if 0
        block of code //comment = =
```

#endif

break your program:

interface file(.h) type definitions and function heaers

implementation file(.c .cc. cpp)

To compile

1. g++ *.cc
2. g++ 1.cc 2.cc

Note: You never compile header files

Note: You never include .cc files

Separate Compilation: Ability to compile individual pieces of a program and then combine them;

g++ main.cc

— compiles

— does not link

By default g++ tries to compile, link of produce executable

Separately compile use -c option to g++ JUST COMPILE

Produces an object file (.o)

binary for compiled code

information about what is available and what is missing

g++ -c main.cc

g++ -c vector.cc

g++ main.o vector.o

links acts as the matchmakers

Global Variables

In abc.h

int globalvar; //declaration and definition

each cc file that includes abc.h has its own globalvar, linker won't link the .o file

In abc.cc

int globalVar;

Since we don't include cc files, we will have one .o file with a globalvar

we need to have a declaration of the global in a header file (without a definition)

In new abc.h

extern int globalvar; //just delcaration

Sperate/example3

Does not compile

In linearAlg.cc

#include "vector.h" — definition of vec

#include "linearAlg.h" 2nd def of vec

Include guard

vector.h

#ifndef __VECTOR_H__

#define __VECTOR_H__

struct ….

#endif

Advice: Always place an include guard

Advice: never put using namespace std in your header file

Advice: Always refer cout as std::cout

cin as std::cin

string as std::string

C++ class

“C++ can put functions inside a struct”

```
struct student{
        int assns, mt, final;
        float grad() {
                return ….
        }
};
student s = {..,..,..};
cout << s.grade();
```

A class is a struct that can contain functions

every struct can have functions

struct actually is a class

Keyword “class” exists; later;

An instance of a class is called an object

S is an object of type student

The function inside a struct (class) is called a member function (method)

In side grade, assns, mt, final to the marks of the object on which grade was called

Lecture 12: Constructors

Function vs Method

A method always has a hidden parameter called “this”

This is a pointer to the object on which the method was called

Student s = {80, 55, 70}; //Style Syntax, compile time constants

Would be nice to be able to write method to initialize objects, such a method is called a constructor.

```
struct Student {
        int assns, mid, final;
        Student(int assns, int mid, int final) { //same as class name, no return type
                this->assns = assns;
                this->mid = mid;
                this->final = final;
        }
};
```

Student billy(60, 70, 80); //construction

Student billy = Student(60, 70, 80);

construct object billy on the stack

Student *billyPtr = new Student(60, 70, 80);

delete billyPtr;

advantages of constructor

+ arbitrary computation

+ default arguments (int assns = 0, int mt = 0, int final = 0)

+ overloading

Student s(60, 70, 80);

Student s1(60, 70); //final default 0

Student s2(60); mt = final = 0;

Student s3(); // function declaration

Every class comes with default (0 argument) construct which calls default construct on ember fields who have construct;

default constructor goes away as soon as you write a constructor of your own

As soon as you write a constructor, you loss c-style initialization

unable to use Vec v = {1, 2};

cannot initialize constants and references inside a constructor body;

steps that occur when an object is create

1. space is allocated (stack/heap)

2. fields are initialized, default constructors

3. constructor body runs

Hijack step 2: member initialization

```
struct MyStuct {
    const int myConst;
    int &myRef;
    MyStruct(int c, int &r) : myConst(c), myRec(r) {
    }
};
```

Fields are initialized in declaration order irrespective of the order in member initialization list

+ only way to initialize constants/referencee

+ same name for field/parameter

+ can be more efficient

```
Student billy(60, 70, 80);
Student body = billy;
```

constructing an object as a copy of another

-copy constructor

-one is available for true

Node *n = …

Node m = *n

nest(other.nextlnext? new Node(*other.next):null)

Lecture 13: Destructor, Assignment Operator

Places where the copy constructor is called

- when we create an object is a copy of another
- when an object is passed by value to a function
- when an object is return from a function

Destructor

To destroy an object, a method called the destructor (dtor) runs.

A dtor has the same name as the class, but prefixed with a tilde(~)

~Node()

A dtor takes no arguments, there is always only one door for a class

You default dtor for free

stack allocated object - object is destroyed when it goes out of scope

heap allocated object - when we call delete on it

The default door calls the dtors on fields that are objects

(note: base types (int, bool stc) and ptrs to objects do not have dtors)

Node *np = new Node(1, new Node(2, new Node(3, NULL)))

np(stack)->1(heap)->2(heap)->3(heap)

1. No deletes - all 3 nodes leak
2. delete np; - 2 and 3 leak, 1 is freed
3. ~Node() { delete next; }

Separate com;ilation of classes

```
struct Node {
        fields
        Node() {}
        Node() {}
};
```

In a .h file put the type definitions including method headers

In the .cc file we implement those methods.

        classes/sparate

        Use Node:: in the cc file - scope resolution operation

```
Student bobby(60, 70, 80);
Student billy = bobby; // copy ctor
Student jane;
jane = bobby;
```

Assignment operator executed, jane already exists

Default operator= does a field for field copy

Custom operator= if there is dynamic memory involved

```
struct Node {
        Node &operator=(const Node &other) {
                if(this == &other) return *this;
                delete next; //may leaks memory if next is pointing something already
                if(*other.next) next = new Node(*other.next); else next = NULL;
                return *this;
        }
}
```

if new fails, next is not assigned, since we deleted next, next is a dangling ptr.

Solution is delay the delete

Node *tmp = next;

Copy and swap idiom

```
struct Node {
        void swap(Node &others) {
                int tdata = data;
                data = other.data;
                other.data = tdata;
                Node *tnext = next;
                next = other.next;
                other.next = tnext;
        }
        Node &operator=(const Node &other) {
                Node tmp = other; // copy actor, stack allocated, deep copy;
                swap(tmp); //tmp has the old data, deleted by stack
                return *this;   //“this” has new values
        }
}
```

Rule of 3

If you need to write a custom version of 1. copy ctor 2. operator= 3. destructor, then you usually need to implement all three

Lecture 14: arrays, const, static

```
struct Node {
        int data;
        Node *next;
        Node(int data):data(data), next(0) {}
};
```

one argument constructor

```
Node n(4);
Node n1 = 4;
```

One argument ctor creates an implicit conversion

```
void foo(Node n);
foo(n);
f00(4);
```

cons: silent conversion

```
istring abc = “hello”;
```

To disable the implicit conversion, use the explicit keyword

```
struct Node {
        explicit Node(int data) {}
};
```

Arrays of Objects

```
struct vec {
        int x, y;
        vec() {} //zero parameter ctor is needed
        vec(int x, int y) : x(x), y(y) {}
};
vec vector[3];
```

```
vec *p = new vec[3];
```

stack

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

heap/stack

- default values to give a default constructor
- write a 0 argument constructor

head/stack: create array of vector ptrs instead of objects

```
vec *vectors[10];
vec **myvectors = new vec * [10];
```

Member functions vs standalone functions

Note: operator= is always implemented as a method.

```
Node &operator=(const Node &other) {}
struct vec {
	int x, y;
	vec operator+(const vec &v) {
		return vec(x + v.x, y + v.y);
	}
	vec operator*(const int k) {
		return vec(x*k, y*k);
	}
	ostream &operator<<(ostream &out) {
		out << x << y << endl;
		return out;
	} // will compile, but v << cout;
};
```

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

operators << and >> always implement as standalone functions

Following operators must be implemented as methods functions

operator=

operator[]

operator()

operator->

operatorT()

const is back!!

```
int foo(const Node n) // cannot modify its fields
```

what methods can I call on const objects

```
struct Student {
        int assns, mid , final;
        mutable int numcalls;
        float grade() const {
                numcalls ++;
                return …..;
        }
};
```

static keyword

static fields: a static field is associated with the class not any object

one field shared by all object

```
struct student {
```

```
        static int numInstances; //declaration
        student(…) .. {
                numInstances ++;
        }
};
```

in cc file

```
int student::numInstances; /define, default init
int student::numInstances=5;
```

static member functions

are associated with the class

you don't need an objet to call this member function

don't have access to "this" parameter

Restrictions:

static member funtions an only call other static functions as access static fields

```
struct student {
        static void print() {
                cout << numInstances;
        }
};
student::print();
```

Lecture 15: Singleton Pattern, Encapsulation

Design Pattern: If you have THIS situation, then THIS programming technique might help.

Singleton Pattern: We have a class C and we want only one instance (object) to be created (throughout the life of the program).

examples:

-database connection

-error log

Finances

Wallet: only have one wallet (singleton)

Expense: many expenses all accessing same wallet.

wallet.h

```
struct wallet {
    static wallet *instance;
    static wallet *getInstance();
    wallet();
    int money();
    void addMoney(int amount);
};
```

wallet.cc

```
wallet *wallet::instance = NULL;
wallet *wallet::getInstance() {
    if(!instance) {
        instance = new Wallet;
        atexit(cleanup);
    }
    return instance;
}
```

```
wallet::wallet() : money(0) {
}
void wallet::addMoney(int amount) {
        money += amount;
}
```

expense.h

```
struct Expense {
        const std::string desc;
        const int amount;
        Wallet *wallet
        expense(std::string desc, int amount);
        void pay();
};
```

expense.cc

```
expense::expense(string desc, int amount) : desc(desc), amount(amount) {
        wallet = wallet::getInstance();
}
void expense::pay() {
        wallet->addMoney(-amount);
        cout << “paying …”;
}
```

problem: we create a wallet on the heap. Never delete it. (Memory leak!!!!)

can’t rely on end of main function to delete instance (wallet);

borrow from c

atexit(functionName) <cstdlib> no arguments, void return

can stack up multiple calls to atexit

```
static void wallet::cleanup() {
        cout << “called clean up”;
        delete instance;
}
```

Tool: valgrind to check memory leak

valgrind ./prog

“all heap block freed”

Encapsulation:

- keep certain implementation details hidden
- object appears as a black bos with exposed interface

```
struct vec {
        vec (int x, int y) : x(x), y(y) {}
        private:
                int x, y;
        public:
                vec operator+ ……
};
```

x and y are now private

```
int main() {
        vec v(1, 2);
        v.x=5; // is not allowed
}
```

default visibility in a struct: public

default visibility should be private, so that it makes the programmer think before making something public

C++ introduces the “class” keyword.

```
class vec {
        int x, y;
        public:
                vec(int x, int y);
};
```

always keep fields private

With private fields, you can provide users with the field values using getters and setters.

```
int vec::getX() { return x; }
int vec::setX(int _x) { x = _x; }
```

Hide implementation details (including fields)

- maintain class invariants
- change implementation

vec : fields private

suppose: don’t want to expose get and set methods

but: want to implement the output operator

Lecture 16: System Modelling

- Vec private x, y fields, no public get or set methods,
- implement operator <<

```
ostream &operator<<(ostream &out, const Vec &v); //cannot access to x, y
```

C++ has a “friend” keyword

```
class Vec {
```

```
    int x, y;
    public:
    friend std::ostream &operator << (std::ostream &, const Vec &c);
};
```

operator << is NOT a member.

advice: have few friends as possible — break encapsulation

System Modelling

Better to design first, then implement

1. abstractions: what classes will I have
2. relationship between classes

Formal technique: UML(unified modelling language)

A UML class is a box

Vec	name of class
- x:integer - y:interger	fields (optional) -private +public
- set x + get x	method (optional)

Relationship 1: Composition

```
Class Vec {
    int x, y, z;
    public:
    vec(int x, int y, int z) : x(x), y(y), z(z) {}
};
class Plane {
    Vec v1, v2;
};
```

Plane p; //does not compile, no default actor for vet

1 provide a default actor

2 create a ctor for Plane which calls the non-default actor for Vec

```
class Plane {
        Vector v1, v2;
        Public:
                Plane() : v1(2, 3, 4), v2(1, 0, 1) {}
};
```

Plane p; //will work

Relationship between Plane and Vec is a “owns a” relationship.

Generally, if A “owns a” B, then

- B has no existence outside A
- if A is destroyed, B is destroyed
- if A is copied, B is copied

car “owns a” wheels

Pond “has a” ducks

```
Class Pond {
        Duck * ducks[20];
};
```

Relationship 1: Aggregation

a catalog has car parts

- parts exist outside a catalog

catalog “has a” parts

Implemented through using pointers to objects inside another object

Relationship 3: Inheritance

track of my collection of books

c++/inheritance/example

```
Class Book{
        string title, author;
        int numPages;
        Public:
        Book (string title, string author, int numPages) ……
};
Class TextBook {
        string title, author;
        int numPage;
        string topic;
};
Class Commic {
        string title, authors;
        int numPages;
        string protng; // protagonist
};
```

In C,

1

```
Union BookType {
        Book *b;
        TextBook *tb;
        Commic *cb;
};
```

BookType myBooks[20];

Bad because you need to track what each element contains;

2

array of void*

- worse than union, keep anything in the array
- suffers from the need for book-keeping information

class Book is unchanged

- testbook is a book
- commic is a book

```
class textbook : public book {
        string topic;
};
class comic: public book {
        string protag;
};
```

textbook/comic inherit title, author, numpages etc

textbook objects have 4 fields

we can call on a textbook object any method that we could call on a book object (public methods);

textbook inherited title, .. etc

Textbook objet cannot access them

book is a bas class (super class)

textbook is a derived class (sub class)

Lecture 17: inheritance, virtual

Private fields are private to that class

Textbook::Textbook() : title(title), author(author) ….//does not work

1. title, author … are private

When an object is create

a. space is allocated

b. superclass part is created

c. default ctors, MIL

d. ctor body

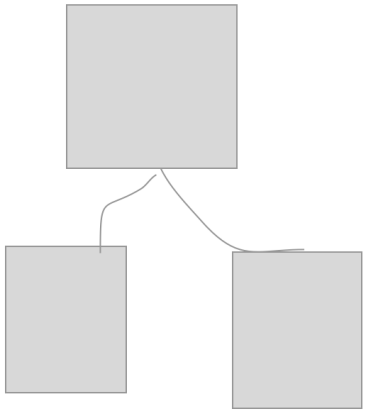
2. No default ctor for book

Textbook::Textbook(…) : book(title, author, …), topic(topic) {}

Lesson: if the superclass does not have default ctor, then call a non-default ctor in the MIL.

```
class Book {
        protected:
        string title author
        …
}
```

When you use p, comic is a book



Book is heavy if > 300

Textbook is heavy if > 500

Comic is heavy if > 30

class Book{

```
        public:
        bool isHeavy() { return numPage > 300; }
}
class Textbook : public Book {
        public:
        bool isHeavy() { return getNumPages() > 500; }
};
```

Book b2 = comin(“comic), “abc”, 40, “adfaf”);

b2.isHeavy(); …… false

If you access an object through a super class variable, then the object is sliced to fit the superclass

comic has been coerced(forced) into a Book

Comic cb(….)

Comic &pcB = &cb

pcB->isHeavy() // true

book *pB = &cb

pB->isHeavy() // false

Lesson: if we access an object through a superclass pointer, the superclass method runs

compiler checks the declared type of the pointer to decide which method to runs

We would like to refer to different Books using Book pointers, and still have the mose “specialized” method called. Use the virtual keyword

```
Class Book {
        public:
```

```
        virtual bool isHeavy() {}
};
```

Virtual Method:

Chooses which method to run based on the actual type of the object at run time

comic cb(…)

book *pb = &cb;

book &rb = cb

pb->isHeavy(); // true

rb->isHeavy(); //true

The process of finding which method to run based on the run time type is called (Dynamic Dispatch) __ small performance penalty

```
book *collection[20];
for(int i = 0; i < 20; i ++; i ++) {
        collection[i]->isHeavy();
}
```

Ability to accommodate multiple types in an abstraction is called Polymorphism (many forms)

Lecture 18: make/ 2d arrays/ abstract/ observes pattern

make： specify a Makefile which lists file dependencies

2D arrays:

int a[10][5];

stack allocated, 10 rows, 5 columns

int *a[10]; is an array containing ptrs to int (postfix before prefix)

```
int **a;
a = new int*[10];


class Student {
        protected:
        int numCourses;
        public:
        virtual int fees() = 0; //method with no implementation, pure virtual
};
class Regular : public Student {
        public:
        int fees() { return 700 * numCourses; }
};
class Coop : public Student {
        public:
        int fees() { return 1300*numCourses; }
};
student *student[400];
for(….) {
        students[i]->fees();
}
```

We do not want to give an implementation for the method Student::fees();

A class with at least one pure virtual method cannot be instantiated, it is an abstract class.

A class that inherits from an abstract class is also abstract unless all inherited pure virtual methods are implemented

A class with no pure virtual methods is called a concrete class (can be instantiated)

uses of abstract classes

- specify the functionality expected from subclasses
- polymorphism
- place common fields in superclass

In UML:

virtual, pure virtual _____ italics

abstract class _____ italics

static _____ underlined

- private, + public #protected

Observer Pattern

Publish-Subscribe model

some entities which publishes/generates data, THE SUBJECT

other entities which subscribe to data to react/consume it, THE OBSERVER

Spreadsheet

Cell: Subject

Plot: observers cells

Horse Races

Subject: publish race result

Observers: betters

```
class Subject {
        Observers *observers[MAX];
        int numObservers;
        public:
```

```
        Subject() : numObservers(0) {}
        bool attach (observer *) {}
        bool detach (observer *) {}
        void notifyObservers() {
                for(int i = 0; i < numObservers; i ++)
                        observers[i]->notify();
        }
        ~Subject() {}
};
```

If you need to make a class abstract but cannot decide on a method, pick the destructor

virtual ~Subject() = 0; // pure virtual

Must implement the destructor, because subclass destructor call superclass destructor

A pure virtual method must be over-ridden in the subclass

Subject::~Subject() {}

Lecture 19: Observer, Decorator, Factory, Template Patterns

```
class Observer {
        public:
        virtual void nofity() = 0;
        virtual ~observer() {}
};
class HorseRace : public subject {
        ifstream stream;
        public:
        …
}
```

```
class Better : public Observer {
        public:
        void notify() ;
};
```

Decorator Pattern

Windoing system (Base window, menu, scroll bar)

```
class coffee {
        public:
        virtual double cost() = 0;
        virtual string desc() = 0;
        virtual ~coffee() {}
};
class espresso : public coffee {
        public:
        double cost() { return 1.0; }
        string desc() { return “full roast espresso”; }
};
class Topping : public coffee {
        protected:
        coffee *c;
        Topping(coffee &c) : c(c) {}
        ~Topping() { delete &c; }
};
class water : public topping {
        public:
```

```
        water(coffee &c) : Topping(c) {}
        double cost { return c.cost() + 0.5; }
}
```

Factory Method Pattern

2 types of enemies

Enemy: Turtle, Bullet, Boss(singleton)

Level: Normal, Castle

```
while ( not Dead ) {
        Enemy *e = l->createEnemy(); // using it as a factory for enemies
}
```

- compartmentalizes enemy creation logic inside levels
- ability to have different logic based on levels

Singleton and factory method pattern often work together

- imagine a Boss enemy, there is only one boss
-
```
while(notDead) {
        Enemy *e = e->createEnemy();
}
```

Template Pattern

put same code in a superclass

```
class Turtle : public Enemy() {
        private:
        void drawHead() { }
        void drawTail() { }
        public:
```

```
        void draw() {
                drawHead();
                drawShell();
                drawTail();
        }
};
```

Non-virtual private methods no subclass can override these

```
class RedTurtle: public Turtle {
        void drawShell() { draw red shell }
};
```

Lecture 20: Templates / Visitor Design Pattern

Templates:

```
class Node {
        int data;
        Node *next;
};
```

Only lets us create linked list of ints

We would have to write new classes to create linked lists of other types

Template: a class parameterized on one or more types

take the int out of the Node class, which will be specified as a type parameters when we create variable

```
template<typename T> class Node {
        T data;
        Node<T> *next;
        public:
```

```
        Node(T data, Node <T> *next) : data(data), next(next) {}
        T getData const { return data; }
        Node <T> *getNext() const { return next; }
}
Node <int> *intList = new Node <int> (1, new Node <int> (2, NULL));
Node <char> *charList = new Node <char> ('a', new Node <char> ('b', NULL));
Node < Node <int> > *linkedListofList; \\not >> for c++ 03 because >> is overloaded
```

STL: Standard Template Library

Dynamic Length Arrays: vector

```
for(int i = 0; i < v.size(); i ++)
        cout << v[i] << endl;// no range checking for i
```

v.at(index);//for range checked access

v.pop_back();//remove last element

Iterators — go through elements of a template class

```
for(vector<int>::iterator i=v.begin(); i != v.end(); i ++)
```

an iterator is an abstraction of a pointer

v.begin(); pointer to the start of the vector

v.end(); pointer ONE PAST the last element of the array

```
for(vector<int>::reverse_iterator i=v.rbegin(); i != v.rend9); i ++);
```

Lookup table: map template parameterized on 2 types

```
#include <map>
using namespace std;
map <<string><int>> m;
```

m[“abc”]=5;

m[“xyz”]=10;

cout << m[“abc”] << endl;

m.erase(“abc”);

m.count(“def”);

iterator returns things in sorted key order

Visitor Design Pattern

Double Dispatch: make a decision on which method to call based on the type of tweet objects

Enemy: turtle, bullet

Weapon: stick, rock

the effect depends on both the weapon and the enemy

Option 1 virtual void Enemy::strike(weapon &w);

Option 2 virtual void Weapon::strike(Enemy &e);

VDP: clever combination of overriding and overloading

```
class Enemy {
        public:
        virtual void strike(weapon &w) = 0;
}
class Turtle: public Enemy {
        public:
        void strike(Weapon &w) {
                w.useOn(*this);
        }
}
```

```
class Bullet: public Enemy {
        public:
        void strike (Weapon &w) {
                w.useOn(*this);
        }
}
```

Lecture 21: Visitor Pattern, Compilation Dependencies

```
class Enemy {
        public:
        virtual void strike(Weapon &w) = 0;
};
class turtle : public Enemy {
        public:
        void strike(Weapon &w) { w.useOn(*this); }
}
class Bullet : public Enemy {
        public:
        void strike(Weapon &w) {
                w.useOn(&this);
        }
}
class Weapon {
        public:
        virtual void useOn(Turtle &t) = 0;
        virtual void useOn(Bullet &b) = 0;
```

```
};
class Stick : public Weapon {
        public:
        void useOn(Turtle &t) {}
        void useOn(Bullet &b) {}
};
class Rock : public Weapon {
        public:
        void useOn(Turtle &t) {}
        void useOn(Bullet &b) {}
};
```

Add a new feature to a hierarchy of classes

- don't want to add code to the existing classes
  - avoid clutter
  - source code not available
- The hierarchy should accept visitors

```
class Book {
        public:
        virtual void accept(BookVisitor &v) {
                v.visit(*this);
        }
};
class TextBook : public Book {
        public:
        void accept(BookVisitor &b) {}
};
```

```
class Comic : public Book {
        public:
        void accept(BookVisitor &v) {
        }
};
class BookVisitor {
        public;
        virtual void visit(Book &b) = 0;
        virtual void visit(TextBook &t) = 0;
        virtual void visit(Comic &c) = 0;
};
```

Catalog of books

Book count books by author

Textbook count books by topic

Comic count books by protagonist

```
class Catalog : public BookVisitor {
        map<string, int> theCat;
        public:
        map<string, int> getCatalog() {return theCat; }
        void visit(Book &b) {theCat[b.getAuthor()] ++;}
        void visit(TextBook &t) {theCat[t.getTopic()] ++;}
        void visit(Comic &c) {theCat[t.getProtag()] ++;}
}
```

There is a cyclic dependancy, but Book visitr.h only needs to know book exists

including book.h is an overkill

forward declare the class

Compilation Dependencies

When to include vs when to forward declare

In a.h

```
class A {};
```

In b.h

```
#include “a.h”
class B : public A {};
```

In c.h

```
#include “c.h”
class C {A myA;};
```

In d.h

```
class A;
class D { A *myA; }
```

client code (clien.cc) includes windows.h

- this creates a compilation dependency
- any change to window.h require compilation client.cc (even the change is to a private field)

Solution: PimP idiom (Pointer to Implementation)

```
class XWindowImpl {
        public:
        Display *d;
        Window w;
        GC gc;
}
```

Contains all previously private fields from XWindow

```
class Window {
        XWindowImpl *pImpl;
        public:
};
```

Lecture 22: Big Three/ Casting

Coupling and Cohesion

Coupling: Degree to which modules depend on each other

Low Coupling: desirable

- prog to an interface
- don’t care about implementation details

High Coupling: depend on low level implementation

- friends, public fields

Cohesion: how related are components within a module

- high cohesion: desirable
- low cohesion: no separation of concerns

Copy ctor

```
Class Book {
        public:
        Book(const Book &other);
};
class TextBook : public Book {
        public:
        //no copy actor implemented
};
TextBook t(,,,);
TextBook copy = t;
To mimic the default TextBook copy ctor
TextBook::TextBook(const TextBook &other) : Book(other), topic(topic) {}

Book &book::operator = (const Book &other) {
        ……
        return *this;
}
Assume I have not implemented TextBook::operator=
        - default one calls Book::operator= then assign TextBook topic field
TextBook &TextBook::operator=(const TextBook &other) {
        Book::operator=(others);
        topic = other.topic;
        return *this;
}
Always perform the operation first on the superclass part of the object
```

TextBook b1(“A”,”Nomair”, 200, “Physics”);

TextBook b2(“B”, “Adam”, 300, “CS”);

Book *pb1 = &b1;

Book *pb2 = &b2;

*pb1 = *pb2; // assign b2 to b1 but through ptrs - superclass ptrs

Book’s operator= is executed

Partial Assignment - to be avoided

make operator= virtual

```
class Book {
        virtual Book &operator= (const Book &other);
};
class TextBook : public Book {
        TextBook &operator=(const TextBook &other);
};
```

Not a valid override - must have same signature

To fix:

```
class TextBook : public Book {
        TextBook &operator=(const Book &other);
};
```

TextBook c();

c = Book(); // what is the topic?

c = Comic(); //worse

Mixed Assignments - to be avoided

Recommendation: make all superclasses abstract

```
class AbstractBook {
        string title, author
        int numPages;
        protected:
        AbstractBook &operator = (const AbstractBook &o);
        public:
        AbstractBook(…): …{};
        virtual ~AbstractBook() = 0;
};
```

In a separate filed, ~AbstractBook() is implemented

```
class NormalBook: public AbstractBook {
        public:
        NormalBook &operator=(const NormalBook &o) {
                AbstractBook::operator=(o);
                return *this;
        }
};
```

assigning through ptrs of a baseclass will not compile

Casting

```
int m = 5;
double d = (double)m;
```

4 casts

1. static_cast

sensible casts

double d = static_cast<double>(m);

Book *bl = new TextBook();

Since we know that b1 points to a TextBook.

it is sensible to use static_case;

TextBook *tp = static_cast<TextBook *>(b1);

- static_cast is unchecked

if your assumption was wrong, you det undefined behaviour.

2. reinterpret_cast “weird conversions”

- allows conversion of prt to a ptr of any type

vec v;

student *s = reinterpret_cast<student *> (&v);

- behaviours is compiler dependent

3. const_cast

- get or take away the const property

void g(int *p) {}

void f(const int *q) {

g(q): //compile error

g(const_cast<int *>(q));

}

- behaviour undefined if q changes p

4. dynamic_cast

vector <book *> myBooks;

book *b = myBooks[s];

- should not use sati_cast because we do not know what b points to

```
Tentatively cast and see if it succeeds
TextBook *t = dynamic_cast<TextBook *> (b);
if b points to a TextBook, after the cast t is a TextBook ptr to the object
if the cast fails, t is set to null
if(t) { t->getTopic(); }
else cout << “not textBook” << endl;
```

Lecture 23: Casting/ Exceptions

dynamic_cast <T*> (expr)

vector <Book*> myBooks;

Book *b=myBookc[i];

- tentative cast, if the cast fails, thefts evals to null

```
Comic *c = dynamic_cast<Comic *>(b);
if(c) c->getProtag();
else cout << “not comic” << endl;
```

To use dynamic_cast, the class hierarchy must have at least one virtual method.

- technical reason:
  - only classes with at least one virtual method have a virtual table (vtable)
  - instances of such classes keep a ptr to this vtable

RTTI: Runtime Type Information

- use dynamic_cast to make decision based on RTTI

```
void whatIsIt(Book *b) {
        if(dynamic_cast<Comic*>(b)) cout << “Comic” << endl;
        else if(dynamic_cast<Textbook*>(b)) cout << “Textbook” << endl;
}
```

- high coupling (poor design)

- brittle (changes to the hierarchy, requires me to search to find all places which use RTTI)
- better use virtual
  - visitor design pattern

Book &b = …;

Comic &c = dynamic_cast<Comic &>(b);

- if b is a ref to a comic, the cast succeeds, c is now a ref to the some comice
- if b isnot a ref to a comic, the cast has failed, what should c be
  - references are never null

Error Handling

In C

- functions return status code
- set a global variable, errno

- Need to keep checking whether an error has occurred.

Error Condition in C++

- dynamic_cast to ref fails
- new fails (no more memory)
- vector's at()method (failed range check)

Code

exceptions/rangeError.cc

If C++

if an error occurs an exception is raised/thrown

default behaviour is terminate the program

if you don't want the program to terminate, then handle/cath the exception

callchain.cc

when an exception is thrown, we hunt the call stack for a suitable handles

- unwind the stack until a catch block is found
- once the handler runs, the statement after the catch executes

Error Recovery can be multi-step process

```
try{…}
catch{SomeErrorType e) {
        partially recovery
        throw someOtherException();
}
```

catches exceptions of type SomeErrorType or any subclass

1. throw;
    - throws the exception that was caught
2. throw e;
    - object coercion might have occurred
    - throw a copy of the exception

advice: use throw;

All built in C++ exceptions inherit from "exception"

C++ does not force use to inherit from exception

```
try {}
catch(…){}
```

Lecture 24: Exceptions/ Exception safety

GoodPractice: create exception classes or use existing ones

```
class BadInput{};
int n;
try {
        if(!(cin >>n)) {
                throw BadInput();
        }
}catch(BadInput &){…}
```

Exceptions can be caught by reference

1. better because it suppresses copies
2. no object slicing occurs

dynamic_cast to a reference fails: bad_cast

new fails(now memory): bad_alloc

accessing an element not in range: out_of_range

operator=

```
        Book *p = new Comic(…);
        Book *p = new Comic;
        *p = *q;
```

1. if operator not virtual, Book::operator= runs — PARTIAL ASSIGNMENT
2. if operator is virtual, we ended up doing MIXED ASSIGNMENT

```
virtual Comic &Comic::operator(const Book &other) {
        Comic &cother = dynamic_cast(Comic&>(other);
        title = cother.title;
…}
```

```
void f() {
        MyClass *p = MyClass;
        MyClass q;
        g();
        delete p;
}
```

Memory leak if g throws!!

Exceptions are meant to be recored from

Our code needs to be exception safe: even if an exception occurs it should not leave a scar (memory leak, dangling ptr)

C++ promise: if an exception occurs, then during stack unwinding, doors for stack allocated objects will be called.

if g throws q does not leak, p does.

```
void f() {
        MyClass *p = new MyClass;
        MyClass q;
        try {
                g();
        }catch(…) {
                delete p;
                throw;
        }
}
```

Tedious

Java — finally clause

Scheme — dynamic_wind

C++ does not need these!!

Advice: prefer allocating objects on the stack

Never ever let the door throw an exception

- dtors are called during stack unwinding (where there is already a live exception)
- if two exceptions are live simultaneously, program terminates

C++ idiom: RAII (Resource Acquisition Is Initialization)

Every resource should be wrapped in a stack-allocated object, whose job is to delete/release the resource

ifstream f("file.txt")

- file is acquired at initialization
- file is released when
  - f goes out of scope -> door runs
  - exception occurs, dotr for f is called during stack unwinding

we can do the exact same thing with dynamic memory i.e. treat it as a resource

C++ standard library provides a class to do exactly this

class auto_ptr<T>

- create an object by giving actor a ptr of type T*
- dtor will delte this ptr

```
void f() {
        auto_ptr<MyClass> P(new MyClass);
        MyClass q;
        g();
}
```

```
class c {}
auto_ptr<c> p (new c);
auto_ptr<c> q = p;
```

if p and q point to the same object, when their doors run it will lead to a double free.

when an auto-ptr is assigned to another, q steals the thing p pointed to, p is set to null.

Exception Safety: 3 levels

1. basic guarantee: if an exception occurs, the program will be in a valid state (no memory leak, no dangling ptrs)
2. strong guarantee: if an exception occurs while executing a function f, then the state of the program is as if f never run
3. No throw guarantee: function never throws an exception; it always achieves its goal.

```
class A{};
class B{};
class C{
        A a;
        B b;
        public:
        void f(){
                a.g();
                b.h();
        }
};
```

g and h have strong guarantee

no strong guarantee because a.g might succeed but h throws

- possible if we can undo changes made by g

Assume g and h have no non-local side effects

- copy and swap

```
f() {
```

```
        A atemp = a;
        B btemp = b;
        atemp.g();
        btemp.h();
        a = atemp;
        b = btemp;
}
```

no strong guarantee because b=btemp; might throw

work on pointers to a & b

Modify the ptrs, swap them in the end.