cin has:

fail “flag” - when read fails - cin.fail() returns true on fail

eof “flag” - when read hits eof -cin.eof() return true on EOF

Note that hitting EOF in a file triggers fail and then EOF

cin - explicitly converts to bool based on state

ie. cin-true-read OK

-false-fail

>> operator:

In C, >> is a bitshift operator

ie. a >> b shift as bits right by by b positions

21 >> 3 = 010101 >> 3

= 010

= 2

>>

LHS: input stream

RHS: variable

eg:

Cin >> x;

(first (second operand variable)

operand

In stream)

- result of expression is another input stream

Cin >> x >> y >> z >> end,

(assign

return cin)

(cin >> y

Return cin)

(cin >> z

Return cin)`

Eg.

Simplify // cin >> x returns cin

int main() {

int i;

while(true) {

if(! (cin >> i)) break; // cleanup

cout << i << endl;

eg. better error checking

int main() {

int i;

while (true) {

if (! (cin>>i)) {

if(cin.eof()) break; //exit on ctrl-D

cin.clear(); //clears the fail bit on cin

cin.ignore(); //ignore the invalid input

} else {

cout << i << endl; // input OK,process it

}

}

}

eg. reading strings

// type std::string (#include <string>

(scope resolution)

int main() {

string s;

cin >> s; //reads a string, skip leading whitespace,

cout << s; (stop reading at whitespace(ie a word))

}

if u want to read whitespace,

getline(cin, s); // s is a string

// breaks on newline

cout << 95 << end; // prints in decimal

print in hex?

cout << hex << 95 << endl; //print 5f

(I/O maipulator)

I/O manipulators:

std::hex - causes output to print in hex

std::dec - ........................................dec

- I/O manipulators stay in effect until you change “back”

- std::endl

-- more in course notes

stream abstractions apply to other sources of data

eg. file access

#include <stdio.h>

int main() {char s[256];

FILE \*file = fopen(“suite.txt”, “r”);

while (true) {

fscanf(file, “%255s”, s);

if feof(file) break;

print( “%\n”, s);

}

fclose(file);

}

C++

# include <iostream>

# include <fstream>

using namespace std;

int main() {

ifstream file {“suite.txt”}; //opens on initial

string s ;

while (file >> s) {

cout << s << endl;

}

} // file out of scope, closes

eg. strings can be stream processed too!

- ie. can use stream to read from /write into strings

- # include <sstream>

- std::istringstream - reading from string

- std:: ostringstream -writing to string

int b = --;

int hi = --;

ostringstream ss;

ss << “Enter a # between” << b << “and” << hi;

string s = ss.str()

eg.convert a string to a number

int n;

while(true) {

cout << “Enter a number” << endl;

string s;

cin >> s;

istringsteam ss(s); // initial string stream with string

if(ss >> n) break; // stop if string contains a number

cont << “I said,”;

}

cout<< “you entered” << n << endl;

eg. revisited

echo all integers, skip non integers

int main() {

string s;

while(cin>>s) {

istringstream ss(s);

int n;

if (ss>>n) cout << n << endl;

}

strings

in C:

-string is an array of characters(char \*, char[]) terminated by \0(null)

- explicitly manage memory - when string grows, need to allocate

more memory

- easy to overwrite \0, causes lots of issues

C++

-strings grow as needed

- not necessarily null-terminated Safer

string s = “Hello”;

string s {“Hello”); // new initial syntax

- even in C++, the literal “Hello” is a C-style string

- the variable s about is a string type that is initialized from a C-style string

string operations

equality, inequality: s1 == s2, s1 != s2

comparison: s1 < s2 (lexographic comparison)

length: s1.length()

concatenation: s3 = s1 + s2

s4 += s5

fetch characters : s3[0] //first

s3[n]

- see course notes

Default function parameters

- ability to supply a default value to a function parameters

void printSuiteFile(string name = “suite.txt”); //default

if stream file(name);

string s;

while (file >> s) cout << s << endl;

}

printSuiteFile(); // use default

printSuiteFile(“suite2.txt”); // alt.

-can have more than one parameter with default

- can mix parameters with/without

-rule?

-parameters with defaults have to be at the end

eg.

void myfunc(string a, string b, string c);

myfunc(“c”)

myfunc(“x”,”y”)

9.27

Overloading(function)

C:

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

bool negBool(bool b) {return !b}

float negFloa(float a) {return -a;}

C++:

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

bool neg(bool b) {return !b;}

float neg(float a) {return -a;}

- function with **different parameter lists** share the same name -----input parameters must differ

-----not just return types

---called overloading

---compiler uses number and type of parameters passed in to decide which function to call

---must differ in number of type of arguments

I/O >> type eg.overloaded function

I/O << type

Structs in C++

struct Node {

int data;

Node \* next;

};

Constants

- invariant values

eg. const int max\_grade:20;

---cannot be changed after initialization

---must be assigned a value

Node n1 {5,nullptr}; // assign values **have to** in order

// do not use NULL, use nullptr to init “empty pointers”

const Node n3 = n1; // copy n1 to n3

Parameter Passing

Recall:

void inc (int n) { n = n+1; }

~~~

int x=5;

inc(x);

cout << x << endl; //5 “Pass-by-value”

“Call-by-value”

Call-by-value

inc() gets a copy of x, and increments the copy

--original x is never touched

-- solution - to modify original, pass a pointer

void inc(int \*n) { \*n = \*n+1;}

...

int x=5;

inc(&x);

cout << x <<endl; //6 x’s address is passed by value, inc()changes the value stored at that address

eg cin >> x;

(function)

C++ has a different pointer type--References

**References** --like an alias to a variable

int y =10;

int &z = y; //z is an Ivalue reference to int y

// like a const pointer int \* const z = y

--references are like constant pointers with auto dereferencing

cout << y << endl; //10

cout << z << endl; //10

z = 12; // now y=12 (not \*z = 12)

int \*p = &y;

int \*q = &z; //same address blc y and z point to same memory address

- z behaves like y; its an alias for y

Things you cannot do with an Ivalue reference

--cannot leave them uninitialized

--must be initialized with something that has an address(ie an Ivalue) Since refs are points

--eg. int &x =3; //NO

int &x = y+z; //NO

int &x=y; //OK y is an Ivalue ie has an address

-- cannot create a pointer to a reference

eg int & \*x; //NO -ref. doesn’t have its own address

int \*& x; //YES -ref. to a pointer is OK

-- cannot create a reference to a reference

eg int && r; (&& -legit)

-- cannot create an array of references

eg. int &r[3] = {x,y,z}; //NO

-- what use are references? function parameters

void inc (int &n) ;

n=n+1;

} --- const pointer to the argument n

--- changes to n affect the original variable

---no pointer dereferencing

int x = 5;

inc(x);

cout << x << endl; // prints 6

----------------------------------------

cin >> x // >> function uses pass-by-reference

*std::istream* & operator >>　(std::istream &in, int &in)

(cin) name cin int x

--pass by value -makes a copy

--pass by reference -- no copy, uses address of original

--------------------------------------------------------------------------

“cost” of different calling mechanisms

int f(int n) {...} // pass by value makes a copy

struct ReallyBig {.....} // Smb (bitmaps)

int f(ReallyBig rb) {.....} // copy, potentially very slow

int g(ReallyBig &rb) {....} // reference, probably very fast

// no new memory required

-- pass by value -original variable cannot be changed(“safe”)

-- pass by ref - original can be changed (“risky”)

int h(const ReallyBig &rb) {...} -- original cannot be changed because of const Keyword

Advice:

-first choice - pass by const ref preferred

--very fast

--safe

--but assumes read-only

- second choice - pass by ref

Pass by value? -single int -OK

-if you want a copy -OK

int f(int &n) {....}

inf g(const int &n) {...}

f(5); // 5 is not an Ivalue (no address)

g(5); // 5 is given a temporary address by g() and will be passed in due to const -OK

Memory allocation

C: int \*p = malloc(... \* sizeof(int));

free(p);

--DO NOT DO THIS IN C++

--use “new” and “delete” --type safe

--less error prone

struct Node {

int data;

Node \*next;

};

Node \*np = new Node; //np is on the stack, Node is on the heap

...

...

delete np;

-- all local variables reside on the stack

-- variables are auto

--deallocated when they go out of scope

--explicitly allocated memory on heap

--remains allocated until delete() called;

--you must delete heap-allocated memory

Array forms

Node \*myNodes = new Node[10]; // array of 10

// nodes on heap

delete [] myNodes

--do not mix up; new-delete

new[] - delete[]

9.29

Node getmeANode() {

Node n; //stack allocated

return n; // return to callers stack-frame

}

int main() {

Node q=getmeANode(); //get a copy

}

--------------------------------------------

Void getmeANode(Node &n) {

n.data=5; //populaters in

}

int main() {

Node n; //stack

getmeANode(n); // populates

}

Node \*getmeANode() {

Node n; //stack

return &n; //return pointer -BAD, crash

-address points to something that no longer exist

}

Node \*gemmeANode() {

Node \*n=new Node; //heap allocated

return n; //return pointer to heap-allocated memory -OK

}

Stack - cost is a copy

heap - manage explicitly ie. someone has to deallocate

-------------------------------------------

Operator Overloading

We can give meaning to operators for new types that we create +,-,... // exist for ints,bools

eg.

struct Vec {

int x,y;

}

-want to add(+) 2 values, mult(\*)by a scale

Vec operator + (const Vec &v1, const Vec &v2) {

Vec v {v1.x + v2.x, v1.y + v2.y};

return v;

}

ie

Vec v1 {1,2}

Vec v2 {3,4}

Vec v3 = v1+v2

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

return {k\*v1.x,k\*v1.y}; // this is ok

} --compiler instantiates a new Vec on the stack --determines what to do based on return type

Vec v5 {3,2};

Vec v6 =2\*v5 //6,4

Vec v7 = v6 \* 12;// order is “wrong”

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

return k\*v; //calls the other operator \*

}

v7=v6\*12 // OK(72,48)

v8=6 \* v2 \* 7

call1 call2

Overloading << and >>

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; //data from 0-100

if(g.theGrade<0) g.theGrade=0;

if(g.theGrade>100) g.theGrade = 100;

return in;

}

cin >> grade

Preprocesser

Transform your program befoe its compiled

#~~~ - preprocessor directive , eg #include

side note: include old headers (c), new naming convention.

eg. #include<stdio.h> -----> #include <cstdio>

old c c++

#define VARIABLE VALUE

- sets a preprocessor variable

-note -no type check

- when it runs, all instances of VARIABLE replaced with VALUE

eg #define MAX 10

int x[MAX]; //transformed to in x[10]

-bad form in c++ -in c++,use const int max = 10, int x[max];

#define FLAG

-sets the variable (values is empty string)

-why? define constants for conditional compilers

eg #define BBOS 1

#define IOS 2

#define OS BBOS//or switch to IOS

....

#if OS == BBOS

long long int Key;

#if IOS == BBOS

long long int Key;

#elif OS == IOS

short int Key;

#endif

----------------------------------------

#if o .... #endif -never true,removed before compilation

-”heavy-duty” comment - removed

eg. DEBUG flag in code - output more stuff when debugging

-first idea -can pass in values on command-line

int main() {

cout << x << endl; // x doesn’t exist?

}

this will not compile like this - need to tell prepocessor to handle x

$g++ -DX=15 define.cc -o define ----D means direct to preprocessor

$./define -----X=15 pass this in

$15

eg. DEBUG flag from command line (debug.cc)

int main() {

#ifdef DEBUG

cout << “setting x = 1” << endl;

#endif

int x = 1;

while (x < 10) {

++x;

#ifdef DEBUG

count << “x is now “ << x << endl;

#endif

}

cout << “x is” << x <<endl;

}

g++ debug.cc -o debug

./debug

x is 9 “normal” DEBUG is not set

g++ DDEBUG debug.cc -o debug

./debug

setting x = 1

x is now 1

x is now 2

...

Separate compilation

--split program into modules with each modules separates

-interface - declaration

-implementation - definition

eg.

//vec.h (interface)

struct Vec {

int x,y;

}

Vec operator + (const Vec &v1, const Vec &v2);

//main

#include <vec.h> //tells main about vec

int main() {

Vec v1;

...

}

//Vec.c(implementation)

#include <vec.h>

Vec operator + (const Vec &v1, const Vec &v2) {

return {v1.x + v2.x,v1.y+v2.y}

}

--recall: you can declare things many times

but define only once- allocates memory

10.4

header- declarations,function prototypes (.h)

#ifndef \_\_HEADER\_H

#define \_\_HEADER\_H

.....//header code

//#endif

source - full implementation (.cc) - .cpp

notes

10.6

struct Node {

int data;

Node \*next;

Node (int data, Node \*next) :data{data},next{next}{}

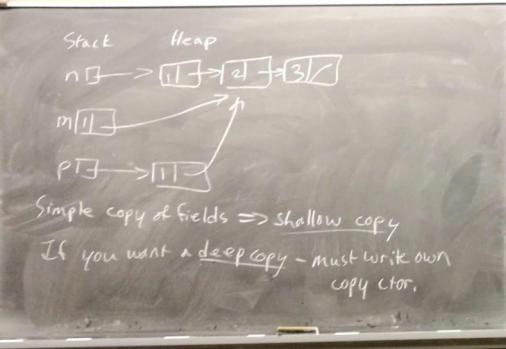
...

};

Node \*n = new Node {1,new Node {2,new Node {3,null ptr}}};

Node m = \*n;

Node \*p = new Node(\*n);



simple copy of fields => shallow copy

if you want a deep copy - must write own copy ctor

struct Node {

...

Node (const Node &other) :

data{other.data}

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

{} calls copy ctor - recursively copies the whole list

};

The copy ctor is called:

1. When an object initializes another object
2. When an object is passed by value
3. When an object is returned by value ((2,3)(\*)-exceptions

-----------------------------------------------------------------------------------

Uniform Initialization

int x= 5; int x(5); int x{5};

string s = “Hello”; string s(“Hello”); string s{“Hello”};

Student billy(60,70,80); Student billy{60,70,80};

Careful - with ctors that can take one argument

struct Node {

...

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

};

Single -arg ctors create implicit conversions

Eg Node n(4);

-but also Node n = 4; implicit conversion from int to Node

int f(Node n) {...}

f(4); //works - 4 implicitly converted to Node

Danger - accidentally pass an int to a fn taking a Node

- compiler will not signal an error

Good idea - disable the implicit conversion

- make the ctor explicit

struct Node {

...

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

{}

};

Node n{4}; //OK

Node n = 4; // Error

Destructors

When an object is destroyed (stack-allocated: goes out of scope

heap-allocated : is deleted)

a method called the destructor (dtor) runs

1. dtor body runs
2. fields dtors are run
3. Space is deallocated

Classes come with a dtor (empty body - just does step2)

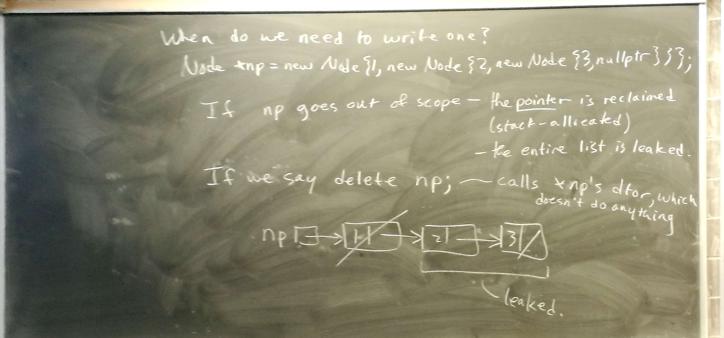
When do we need to write one?

Node \*np = new Node {1, new Node {2, new Node {3, nullptr}}};

If np\*goes out of scope - the pointer is reclaimed(stack-allocated)

- the entire list is leaked

If we say delete np; calls \*np’s dtor, which doesn’t do anything



Write a dtor to ensure the entire list is freed

struct Node {

...

~Node() { delete next;}

}; recursively calls \*next’s dtor, free the whole list

Now, delete np; frees the whole list

Copy Assignment Operator

Student billy {60,70,80};

Student jane = billy; //copy ctor

Student anne; //default ctor

anne = billy, // copy, but not construction

copy assinment operator - uses compiler - supplied default

May need to write your own:

struct Node {

...

//in c we can be a=b=c=4;

//so our operator is written so that cascading works

Node & operator = (const Node &other) {

data = other.data;

delete next;

next = other.next? new Node {\*other.next} :nullptr;

return \*this;

}

}; //DANGEROUS

Why? Node n {1, new Node {2, new Node {3, nullptr}}};

n = n; -deletes n then tries to copy n to n

-undefined behaviors

\*p = \*q

a[i] = a[j]

When writing operator=, ALWAYS be wary of self-assignment;

struct Node {

...

Node &operator= (const Node &other) {

if (this == &other) return this;

data = other.data;

delete next;

next = other.next? new Node {\*other.next}: nullptr;

return \*this; list is corrupted if new fails

}

};

Better:

Node &operator= (const Node &other) {

if (this == &other) return \*this;

Node \*tmp = next;

next = other.next? new Node {\*other.next} : nullptr;

data = other.data;

delete tmp;

return \*this;

}; // if new fail, Node is still in a valid state

Alternative: copy + swap idiom

#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; //tmp - copy of other...

swap(tmp); // me - copy of other

tmp - my old data

return \*this;

} // tmp stack-allocated

}; - destroyed on return

- reclaims my old data

10.13

Rvalue Rvalue Reference

Recall: -an Ivalue is anything with an address

-an Ivalue reference (&) is like a const. ptr with auto-deref = always initialized to an Ivalue

Node n{1, new Node{2,nullptr}};

Node m = n; //copy ctor

Node mz;

mz = n;//copy assignment operator

Node plusOne(Node n) {

for (Node \*p = &n; p; p = p->next)) {

++ p->data;

}

return n;

};

Node m3 = plusOne(n);//copy ctor

What is “other” here?

-reference to what?

-compiler creates a temporary object to hold the result of plusOne.

-other is a reference to this temporary

-copy ctor deep copies the data from this temporary

But - the temp is going to be discarded, as soon as Node m3 = plusOne(n); is done

- wasteful to copy data from a temp-why not just steal it instead? - save the cost of a copy

- Need to be able to tell whether other is a reference to a temporary or standalone object.

C++ - rvalue reference Node&& is a reference to a temporary object(rvalue) of type Node

Version of the ctor that takes a Node &&:

struct Node {

...

Node (Node &&other): // move ctor

data{other.data}; what should it do?

next{other.next} { - steal other’s data

other.next = nullptr; //so the list is not lost when other is destroyed

}

}

Similarly: Node m;

m = plusOne(n); // Assignment form temporary

Move assignment operator:

struct Node {

...

Node &operator= (Node && other) { // steal other’s data

swap(other);

return \*this; //Destroy my old data

} //temp will be destroyed

} // and take our old data with it // swap without copy

If you don’t define move ctor /assignment, the copy versions will be used

If you do define them, they replace all calls to copy ctor /assignment when the argument is a temporary(rvalue)

Vec makeAVec() {

return {0,0}; //invokes basic ctor

}

Vec v = makeAVec(); // What’s this? copy ctor?move ctor?

Not sure! In g++: just the basic ctor

-no copy ctor, no move ctor

In some circumstances, the compiler is allowed to skip calling copy/move ctors (but doesn’t have to)

In our example: make A Vec writes its result ({0,0}) directly into the space occupied by V in the caller, rather than copy/move later

Example:void doSomething(Vec v) {...}

copy/move ctor

doSomething (makeAVec());

result of makeAVec() written directly into the parameter - no move/copy

This is allowed, even if dropping ctor calls would change the behavior of the program(eg if the ctor print something)

You are not expected to know exactly when, when copy/move elision is allowed-just that it is a possibility

If you need all of the ctors to run:

g++14 -fno-elide-constructors

but this can slow down your program considerably

In summary: Rule of 5(Big 5)

If you need to customize any one of

\*copy ctor

\*copy assignment

\*dtor

\*move ctor

\*move assignment

then you usually need to customize all five

Notice: operator = is a member fn, not a standalone fn

When an operator is defined as a member fn, this plays the role of the first arg.

struct Vec{

struct x,y;

...

Vec operator+(const Vec &other) {

return {x+other.x, y+other.y};

}

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

} // //implement v\*k

To implement k\*v - can’t be a member fn - first arg is not Vec

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

return v\*k;

}

I/O operator:

struct Vec {

...

ostream &operator << (ostream &out) {

return out << x << ‘ ‘ << y;

}

};

What’s wrong? - Makes Vec the first argument, so use as v << cout;

So define operator << , >> as standalones

Certain operators must be members:

\*operator =

\*operator []

\*operator ->

\*operator()

\*operator T (where T is a type)

Separate Compilation classes

Node.h

#.....

struct Node {

int data;

Node \*next;

explicit Node (int data, Node \*next = nullptr)

#endif

Node.cc

#include “Node.h”

Node::Node(int data, Node \*next): data{data},next{next} {}

bool Node::hasNext() {return next != nullptr;}

:: - called the scope resolution operator

Node :: ---- means ----- in the context of struct Node

:: like . where LHS is a class (or nonespace), not an object

10.18

Arrays of Objects

struct Vec {

int x,y;

Vec(int x,int y);

};

Vec \*vp = new Vec[5];// X(false)

Vec moreVecs[3];// X(false) These want to call the default ctor on each item

you can’t initialized the array items - no default ctor

Options: 1) Provide a default ctor

2) For stack arrays

Vec moreVecs[] = {{0,0},{1,3},{2,4}};

1. For heap arrays: - create an array of ptrs

Vec \*\*vp = new Vec \*[5];

vp[0] = new Vec {0,0};

vp[1] = new Vec {1,3};

...

for (int i = 0; i < 5; ++i) {

delete [] vp;

---------------------------------------------------------

Const Objects

int f(const Node &n) {...}

const objects arise often

What is a const object? Can’t change fields

Can we call methods on a const object?

Issue: the method may modify fields, violate const

A: Yes- we can call methods that promise not to change fields

Eg: struct student {

int assns,mt,final;

float grade() const;

}; this method doesn’t modify fields

Compiler checks that const methods don’t modify fields

Only const methods can be called on const objects

New Consider

want to collect usage stats on Student

Objects:

struct Student {

...

int numMethodCalls = 0;

float grade() const { can no longer be const

++ numMethodCalls; [now can’t call grade on const students]

return ----- ;

}

};

But mutating numMethodCalls affects only the physical constnes s of Student objects, not the logical constness

Want to be able to update numMehtodCalls, even if the object is constant. -delete the field mutable.

struct Student {

...

mutable int numMethodCalls = 0;

float grade() const {

++ numMethodCalls;

return ------- ;

}

}; mutable fields can be changed, even if the object is const.

Static Fields + Methods

numMethodCalls tracks #of method calls for a particular object

What if we want to track #of methods calls over all objects?

Or what if we want to track how many students are created?

Static members - associated with the class itself, not with any specific instance(object)

struct Student {

...

static int numInstances;

Student (int assns, int mt, int final) : ------- {

++ numInstances;

}

};

int Student::numInstances = 0; // in .cc file

Static fields must be defined external to the class

static member fns:

-don’t depend on the specific instance (no this parameter)

-can only access static fields & call other static methods

struct Student{

...

static int numMethods;

...

static void print NumInstances() {

cout << numInstances << endl;

}

};

Student billy {60,70,80};

Student jane{70,80,90};

Student::print NumInstances(); // 2

Invariants + Encapsulation

struct Node {

int data;

Node \*next;

Node(int data, Node \*next);

~Node() {delete next;}

};

Node n1 {1,new Node {2, nullptr}};

Node n2 {3, nullptr};

Node n3 {4，&n2};

What happens when these go out of scope?

n1 -dtor runs, entire list is deleted OK

n2,n3 go out of scope - n3 attempts to delete n2, but n2 is on the stack, not the heap!

undefined behavior

Node relies on an assumption for its proper operation:

that next is either nullptr or allocated by new.

This is an invariant - statement that hold true

- on which Node relies

- But we can’t guarantee the invariant - can’t trust the user to use Node properly

-Can’t enforce any invariants - cant control the user

Eg stack: In invariant - last item pushed is first item popped.

- but not if the client can rearrange the underlying data

Hard to reason about programs if you can’t rely on invariants.

To enforce invariants - We introduce encapsulation - want clients to treat objs as black boxes - capsules

- seal away implementation; interact only via provided methods

- abstraction - regain control over our objs,

struct Vec {

Vec(int x, int y); also public by default

private:

int x,y; can’t be accessed outside struct Vec

public:

Vec operator +(----); anyone can access

};

In general - want fields to be private; only methods should be public

Better to have default visibility be private

Switch from struct to class

class Vec {

int x,y;

public :

Vec(int x, int y);

Vec Operator + (----);

...

};

Difference between class & struct = default visibility:

public in struct

private in class

Fix the linked list class;

// list.h

class List{

struct Node; // private Nested class

- only accessible within class List

Node \* theList = nullptr;

public:

void addToFront(int n);

int ith(int j) const;

~List() {delete theList;}

};

//list.cc

#include “list.h”

struct List::Node {

int data;

Node \*next;

Node(-------) {}

~ Node() {delete next;}

};

void List::addToFront(int n) {

theList= new Node {n, theList}；

}

void List::ith (int i) const {

Node \*cur = theList;

for (int n = 0; n < &&cur; ++n,cur=cur->next);

return cur->data;

}

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

Only List can create/manipulate Node objects

so Can guarantee the invariant that next is always nullptr or allocated by new

But - now we can’t traverse the list from node to node as we would a linked list

-Repeatedly calling ith to access the whole list

-O(n²) time

-But we can’t expose the nodes or we lose encapsulation

10.20

SE Topic: Design Patterns

-centain problems arise frequently and have well-studied solutions

-keep track of these of use in similar situations

Design Pattern - If you have problem X, technique Y may solve it

Sol’n: Iterator Pattern

- Create a class that manages access to nodes

-abstraction of a ptr

- walk the list without exposing the actual ptrs

for (int \*p = a; p !=a+n; ++p) }

... \*p ...

}

class List {

struct Node ;

Node \*theList = nullptr;

public:

class Iterator{

Node \*p;

public:

explicit Iterator(Node \*p):p{p}{}

int &operator \*() {return p->data;}

Iterator &operator ++() { p = p->next; return \*this;}

bool operator == {const Iterator &other) { return p == other.p;}

bool operator != (const Iterator &other) {return !(\*this == other); {

};//Iterator

Iterator begin() { return Iterator {theList};}

Iterator end() {return Iterator(nullptr)};

...

};

Client

int main() {

List l;

l.addToFront(1);

l.addToFront(2);

l.addToFront(3);

for(List::Iterator it = l.begin(); it!=l.end();++it){

cout << \*it << endl;

}

}

Shortcut: automatic type deduction

auto x=y;

automatically gives x the same type as y

for(auto it = l.begin(); it!=l.end();++it) {

cout << \*it << endl;

}

Shorter cut - range-based for loop

for(auto n : l) {

cout << n << endl;

}

Available for any class with

\* methods begin + end that produce iterator

\* the iterator must support !=, prefix ++,unary \*

If you want to modify the list elements (or save copying)

for (auto &n : l) {

++n;

}

Iterators shall return

Encapsulation ctd.

List client can create iterators directly:

auto it = List::Iterator {nullptr};

violates encapsulation - client should be using begin/end

we could - make Iterator’s ctor private.

- then client can’t call List::Iterator

- but then neither can List

Sol’n: - give List privileged access to Iterator

- make it a friend

class List {

...

public:

class Iterator {

Node \*p;

explicit Iterator (Node \*p);

public:

friend class List; // List has access to all members of Iterator

};

};

Now , List can still create iterators but client can only create iterators via begin/end

Advice - Give your classes as few friends as possible - weakens encapsulation

-------

Once again - keep fields private

If you want the client access to fields - use accessor & mutator methods instead

class Vec {

int x,y;

public:

int getX() const {return x;} // accessor

void setY(int newY) { y = newY; }//mutator

};

---------- What about operator <<?

- needs x & y, but can’t be a member

-if no getX, getY defined - make operator << a friend function

class Vec {

...

public:

...

friend std::ostream & operator <<(std:: ostream &out,const Vec &v);

};

.cc

ostream &operator <<(ostream &out, const Vec &v) {

return out << v.x << ‘ ‘ << v.y;

}

Tool topic : make

Separate compilation : g++14-c list.cc

g++14-c node.cc

g++14-c iter.cc

g++14-c main.cc

g++14-c list.o node.o iter.o main.o - o myprog

Why do we do this? So we don’t recompile files that haven’t changed.

How do you keep track of what’s changed + what hasn’t.

Let Linux help - with make

Create a Makefile that says which files depend on which other files

myprog: main.o list.o node.o iter.o // myprog depends on these

g++-5 main.o list.o node.o iter.o -o myprog // there is a TAB before g++

list.o: list.cc list.h node.h

g++-5 -std=c++14 -c list.cc

node.o: node.cc node.h list.h

g++-5 -std=c++14 -c node.cc

etc.

Then,from the command line:

“make”

Now just change iter.cc

and type “make” - compiles iter.cc

- relinks main

10.25

Command make - builds the first target (myprogram) in the Makefile

- What does my program depend on?

-- recursively build those, if necessary

-- rebuild myprogram, if necessary

iter.cc changes - now **newer** than iter.o

thus rebuild iter.o

now iter.o newer than myprogram

thus rebuild myprogram

Can also build specific targets,eg make node.o

Common: a target clean: -removes all binaries

-

-

.PHONY: clean

clean: rm \*.o myprogram

to do a full rebuild: make clean

make

-------------------------------------------------

Generalize with variables

CXX = g++-5 (compiler’s name)

CXXFLAGS = -std=c++14 -Wall turn on warnings

eg iter.o:iter.cc list.h

${CXX} ${CXXFLAGS} -c iter.cc

shortcut: for any rule of the form

x.o:x.cc a.h b.h....

can leave out the build command

- make guesses that you want ${CXX} ${CXXFLAGS} -c x.cc -o x.o

Itardest part of writing Makefiles

-dependencies

- and maintaining them if they change

Can get help from g++

g++-5 -std=c++14 -MMD -c iter.c

iter.d

iter.o: iter.cc list.h node.h

Now just include these into the Makefile

.

.

.

CXXFLAGS = -std=c++14 -Wall -MMD

OBJECTS = main.o list.o iter.o node.o

DEPENDS = ${OBJECTS: .o=.d}

....

-include ${DEPENDS}

System Modelling

Building an OO system -

-identify abstractions

-formalize relationships among item

Helpful to map these out

Popular Standard: UML(Unified Modelling Language)

Modelling a class

|  |  |
| --- | --- |
| Name | Vec |
| Field(optional) | -x: Integer  -y: Integer |
| Methods(optional) | +getX:Integer  +getY:Integer |

Visibility: - private

+ public

Relationship: Composition of Classes

class Vec {

int x,y;

public:

Vec{int x, int y};

};

Two Vecs define a basis

class Basis {

Vec v1,v2;

};

Basis b;// -can’t initialize v1,v2

-no default ctor for Vec

class Basis {

Vec v1,v2;

public:

Basis():v1{1,0},v2{0,1}{}

};

Embedding one obj(Vec) inside another(Basis) is called composition

Relationship betwwen Basis & Vec called “owns-a”

A Basis “owns” two Vec Objects.

If A “owns a” B, then typically:

-B has no identity outside A (B is a part of A)

-If A is destroyed, B is destroyed

-If A is copied, B is copied (deep copies)

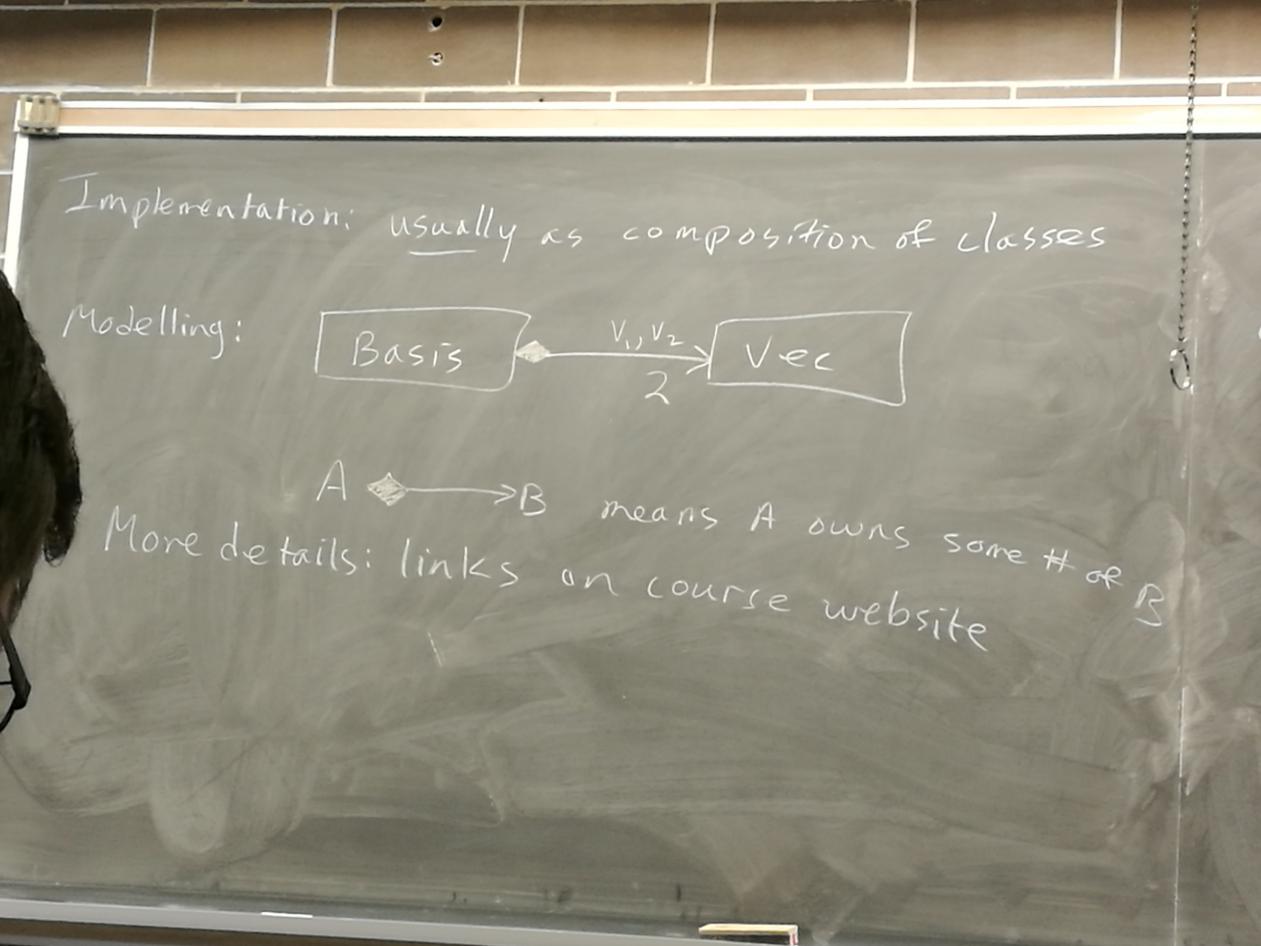
Eg - A car owns 4 wheels -a wheel is part of a car

-destroy the car => destroy the wheels

- copy the car => copy the wheels

Implementation: usually as composition of classes

Modelling:



Aggregation

Compare car parts in a car (“owns a”) Vs. car parts in a catalogue

The catalogue contains parts, but the parts have an independent existence

This is a “has a” relationship(aggregation)

If A “has a” B, then typically:

- B has an existence a part from its association with A

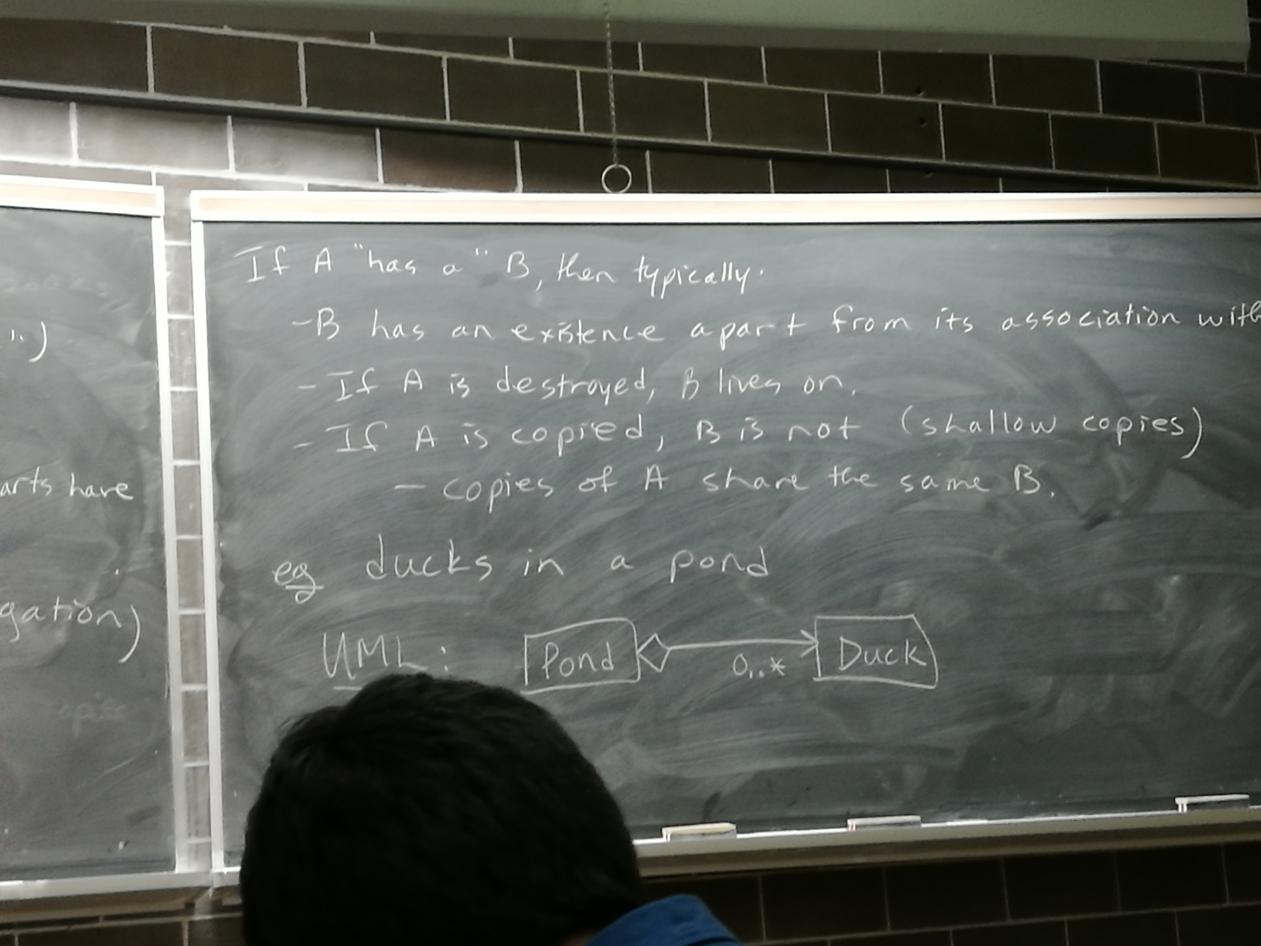
- If A is destroyed, B lives on

- If A is copied, B is not(shallow copies)

- copies of A share the same B

eg ducks in a pond

UML:

[]

Typically Implementation: pointer fields

class Pond {

Duck \*ducks[maxDucks];

};

Specialization / Generalization(inheritance)

Suppose you want to track your collection of books

class Book {

string title,author;

int numPages;

}

class Text{

string title,author;

int numPages;

string topic;

};

class Comic {

string title,author;

int numPages;

string hero;

};

Does not capture the relationships among these classes

How do you create an array or list that contains a mixture of these?

In C++ - inheritance

class Book {

string title,author;

int numPages;

public:

Book(-----);

};

Base class or superclass

class Text:public Book{

string topic;

public:

Text(----);

};

class Comic: public Book{

string hero;

public:

Comic(----);

};

Derived classes or subclasses

Derived classes inherit field + methods form the base class

Text & Comic get title,author,numPages fileds

Any method that can be called on Book, can be called on Text,Comic

Who can see those members?

title,etc private in Book- outsiders can’t see them

Can Text,Comic see them? No, even subclasses can’t see them

How do we initialize Text?

class Text:public Book{

...

public:

Text(string title, string auther, int numPages, string topic): title{title},author{author},

numPages{numPages}, topic{topic}{}

}; WRONG!

Wrong for 2 reasons:

1. title,etc not settable in Text’s MIL
2. when an object is created:

1)Space is allocated

\*2)Superclass part is constructed -Can’t- no default ctor for Book

3)Fields constructed

4)ctor body runs

class Text:public Book{

...

public:

Text(string title,string author, int numPages,string topic): Book{title,author,numPages},

topic{topic}{}

}; //RIGHT!

If superclass has no default ctor, then the subclass must invoke a superclass ctor in its MIL

10.27

Recall

class Text: public Book {

....

public:

Text(string title, string author,int numPages,string topic) : Book{title,author,numPages}; topic{topic}{}

};

If you want to give the subclass access to certain members - use protected visibility

class Book{

protected:

string title,author; |

int numPages; | //accessible to Book and its subclasses

public:

....

};

class Text:public Book {

string topic;

public:

...

void addAuthor(string auth) {author += auth;}

};

Not a good idea to give subclasses unlimited access to fields

Better: private fields with protected accessors

class Book{

string title,author;

int numPages;

protected:

string getTitle() const;

void setAuthor(string auth);

public:

Book(----);

bool isItHeavy() const;

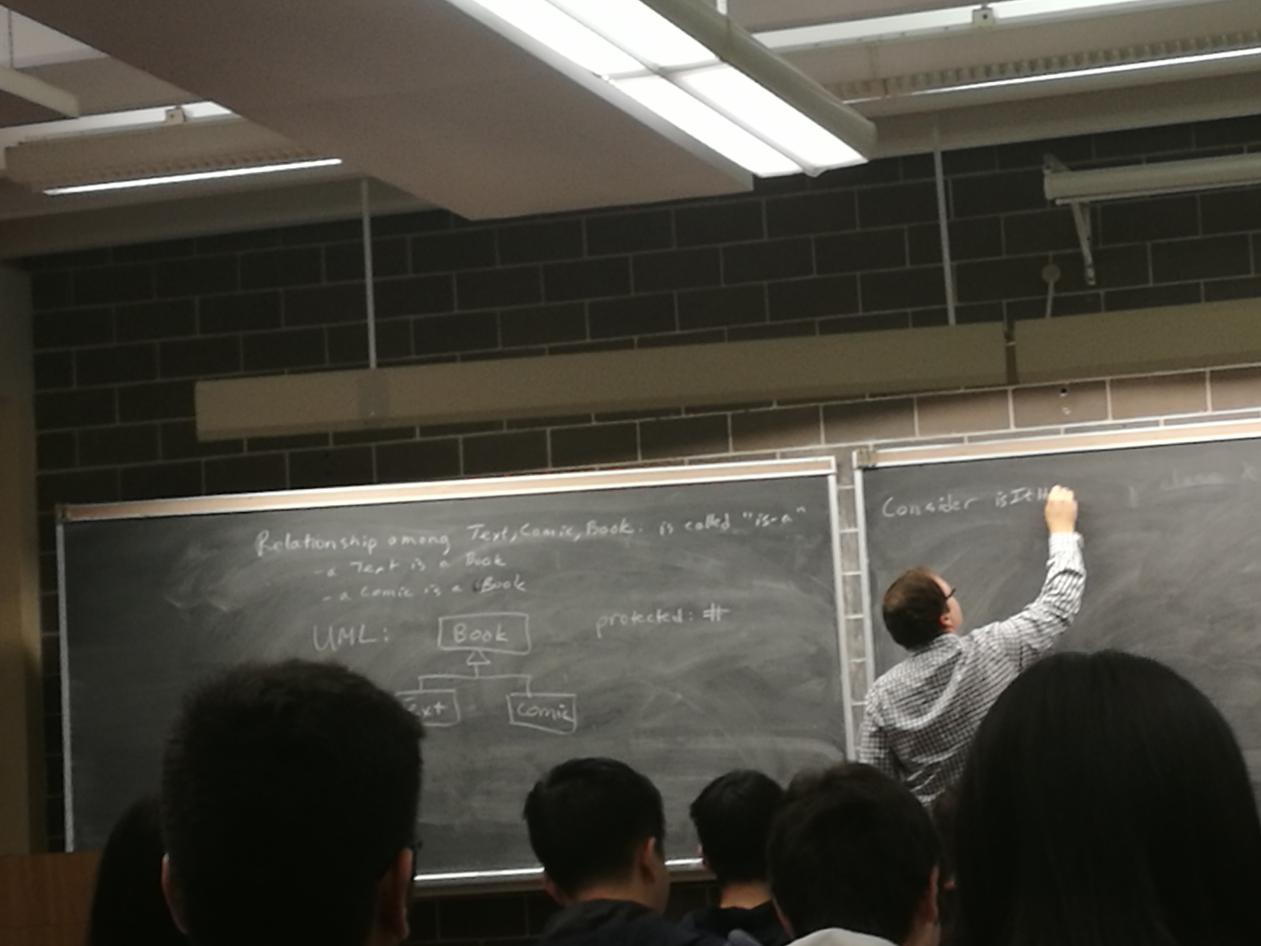
};

Relationship among Text,Comic,Book: is called “is-a”

-a Text is a Book

- a Comic is a Book

UML

[]

Consider isItHeavy - when is a book heavy?

- for ordinary Books - > 200 Pages

-for Texts- > 500 pages;

- for Comics >30 pages

class Book {

...

protected:

int numPages;

public:

bool isItHeavy() const {

return numPages >200;}

};

class Comic: public Book {

...

public:

bool isItHeavy() const {

return numPages > 30;}

};

Text-exercise

Book b {“A small book”,”...”,50};

Comic c {“A big comic”,”...”,40,”Archie”};

cout << b.isItHeavy() //false

<< c.isItHeavy() //false

Since a Comic is a Book,we can do this

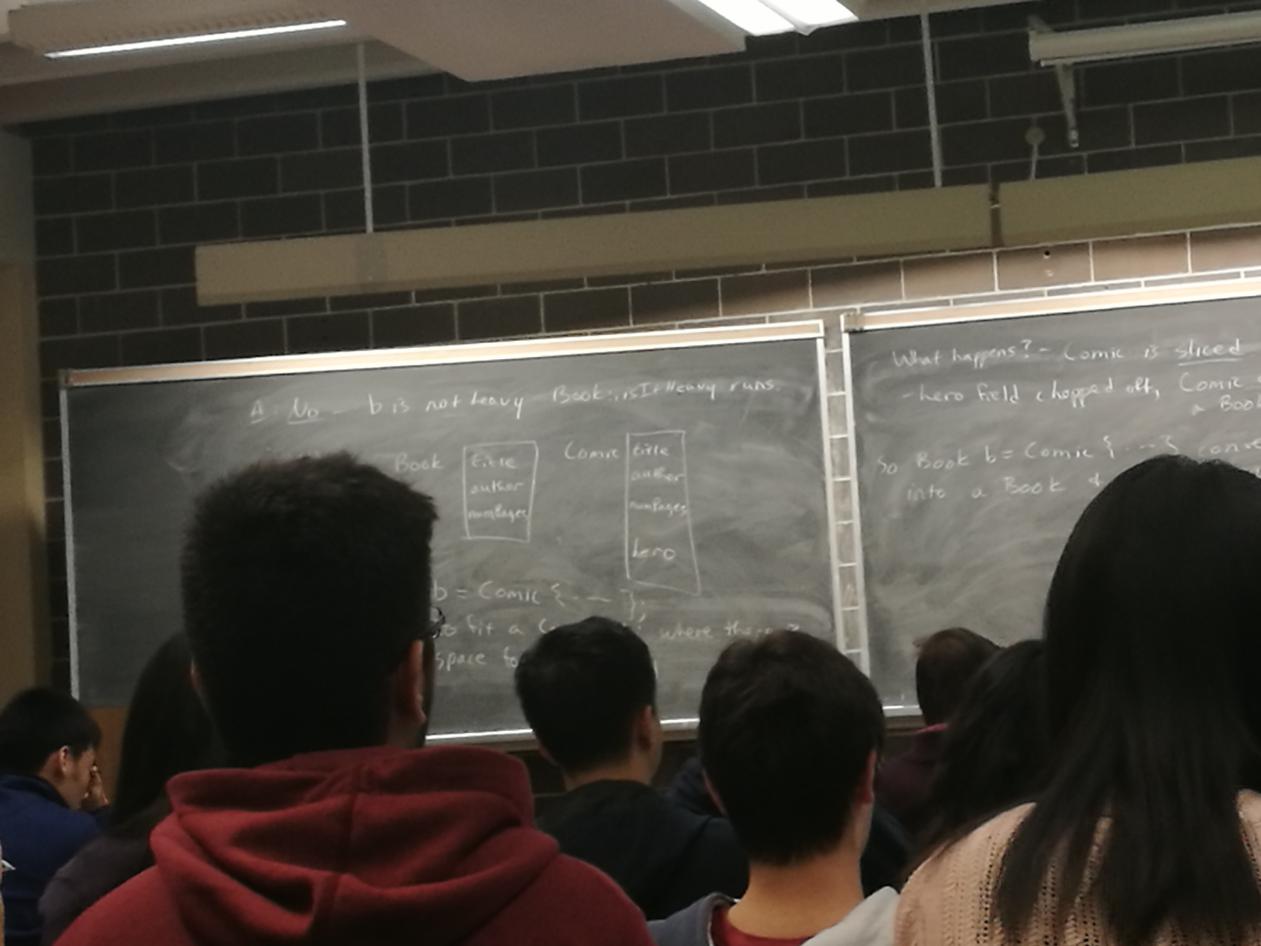
Book b = Comic {“----”,”----”,40,”------”};

Q: Is b heavy? b.isItHeavy() -- true or false?

Which isItHeavy runs? Book::isItHeavy or Comic::isItHeavy?

A:No. b is not heavy Book::isItHeavy runs

Why?

[]

Book b = Comic {----};

-tries to fit a Comic obj where there is only space for a Book obj

What happens? - Comic is sliced

- hero field chopped off, Comic covered into a Book

So Book b = Comic {...} converts the Comic into a Book & Book::isItHeavy runs

When accessing objects through pointers,slicing is unnecessary and doesn’t happen

Comic c { \_, \_,40,\_};

Book \*pb = &c;

Comic \*pc = &c;

cout << pc->isItHeavy() // true

<< pb->isItHeavy() // false (no slicing here)

-- still Book:isItHeavy runs.

Same object behaves differently depending on what type of ptr points at it.

Compiler uses the type of the pointer (or reference) to decide which isItHeavy to run - does not consider the actual type of the object

Means a Comic is only a comic when pointed at by a Comic ptr - probably not what we want

How do we make a Comic act like a Comic,even when pointed at by a Book ptr?

Declare the method virtual

class Book {

...

public:

Book (----);

Virtual bool isItHeavy() const {...}

};

class Comic: public Book{

...

public:

bool isItHeavy() const override {...}

};

Comic c {\_,\_,40,\_};

Book \*pb = &c;

Comic \*pc = &c;

Book &rb = c;

cout << pb->isItHeavy() // true

<< pc->isItHeavy() // true

<< rb.isItHeavy() // true

Comic::isItHeavy runs in all 3 cases

virtual method \_ choose which class method to run, based on the actual typed the object at runtime

eg.

my Book collection

Book \*myBooks[20],

...

for (int i=0; r < 20;++i) {

cout << myBooks[i]->isItHeavy() << endl;

} uses Book::isItHeavy for Books

Comic:: Comics

Text:: Texts

Accommodates multiple types under one abstraction polymorphism (“many forms”)

Destructor revisited

class X {

int \*x;

public:

X(int n): x {new int [n]} {}

~X() {delete [] x;}

};

class Y:public X{

int \*y;

public:

Y(int m,int n): X{m}, y{new int [n]}{}

~Y() {delete [] y};

};

X \*myX = new Y{10,20};

delete myX; -leaks! Why?

calls ~X, but not ~Y

So only x,but not y, is freed

How do we ensure that deletion through a superclass or will call the subclass dtor? - Make the dtor virtual.

calss X {

...

public:

...

virtual ~X() {delete [] x;}

};

ALWAYS - make the dtor virtual in classes that are meant to have subclasses,

declare it final.

class Y final:public X{

...

};

11.1

Pure Virtual Methods + Abstract Classes

class Student {

protected:

int numcourses;

public:

virtual int fees();

};

2 kinds of Student Regular + co-op

class Regular: public Student {

public:

int fees() override;

reg Students’ fees

};

class Coop: public Student {

public:

int fees() override;

coop students’ fees

};

What should we put for Student fees?

Not sure- every student should be either regular or co-op

Can explicitly give Student::fees? No implementation

class Student {

...

public: method has no implementation(\*)

virtual int fees() = 0; // Called a pure virtual method

A class with a pure virtual method cannot be instantiated:

Student s; // false!!!

-called an abstract class - purpose is to organize subclasses

Subclasses of an abstract class are also abstract, unless they implement all pure virtual method

Classes that aren’t abstract are called concrete

UML: Virtual + pure virtual methods: Italics

Abstract classes: class name in italics

Static - underline

------------------------------------------------------------------

Templates

Class List {

struct Node;

Node \*theList;

};

struct List::Node {  
 int data;

Node \*next;

};

What if you want to store something else?

Whole new class?

OR a template

- a class parameterized by a type

template <typename T>

class Stack {

int size;

int cap;

T \*contents;

public:

Stack(){...}

void push(T x) {...}

T top() {...}

void pop() {...}

};

template <typename T>

class List {

struct Node {

T data;

Node\*next;

};

Node \* theList;

public:

class Iterator{

Node \*p;

..

public:

T &operator\*() {return p->data;}

...

}; ...

...

T ith (int i) {...}

void addToFront(T n) {...}

};

Client: List<int> l1;

List<List<int>> l2;

l1.addToFront(3);

l2.addToFront(l1);

for (List<int>::Iterator it = l1.begin(); it != l1.endl;++it) {

cout << \*it << endl;

}

or

for (auto n : l1) {

cout << n << endl;

}

The standard Template Library (STL)

Large #of useful templates

Eg dynamic-length arrays vector

#include <vector>

std::vector<int> v{4,5}; //4,5 //std::initializer\_list <int>

std::vector<int>w(4,5); // 5,5,5,5

v.emplace\_back(6);

v.emplace\_back(7);; // 4,5,6,7

Looping:

for (int i = 0; i<v.size();++i) {

cout<<v[i]<<endl;

} auto

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

cout << \*it <<endl;

}

OR

for(auto n : v) {

cout << n << endl;

}

To iterate in reverse: auto

for(vector<int>::reverse\_iterator it = v.rbegin(); it != rend(); ++it) {

cout << \*it <<endl;

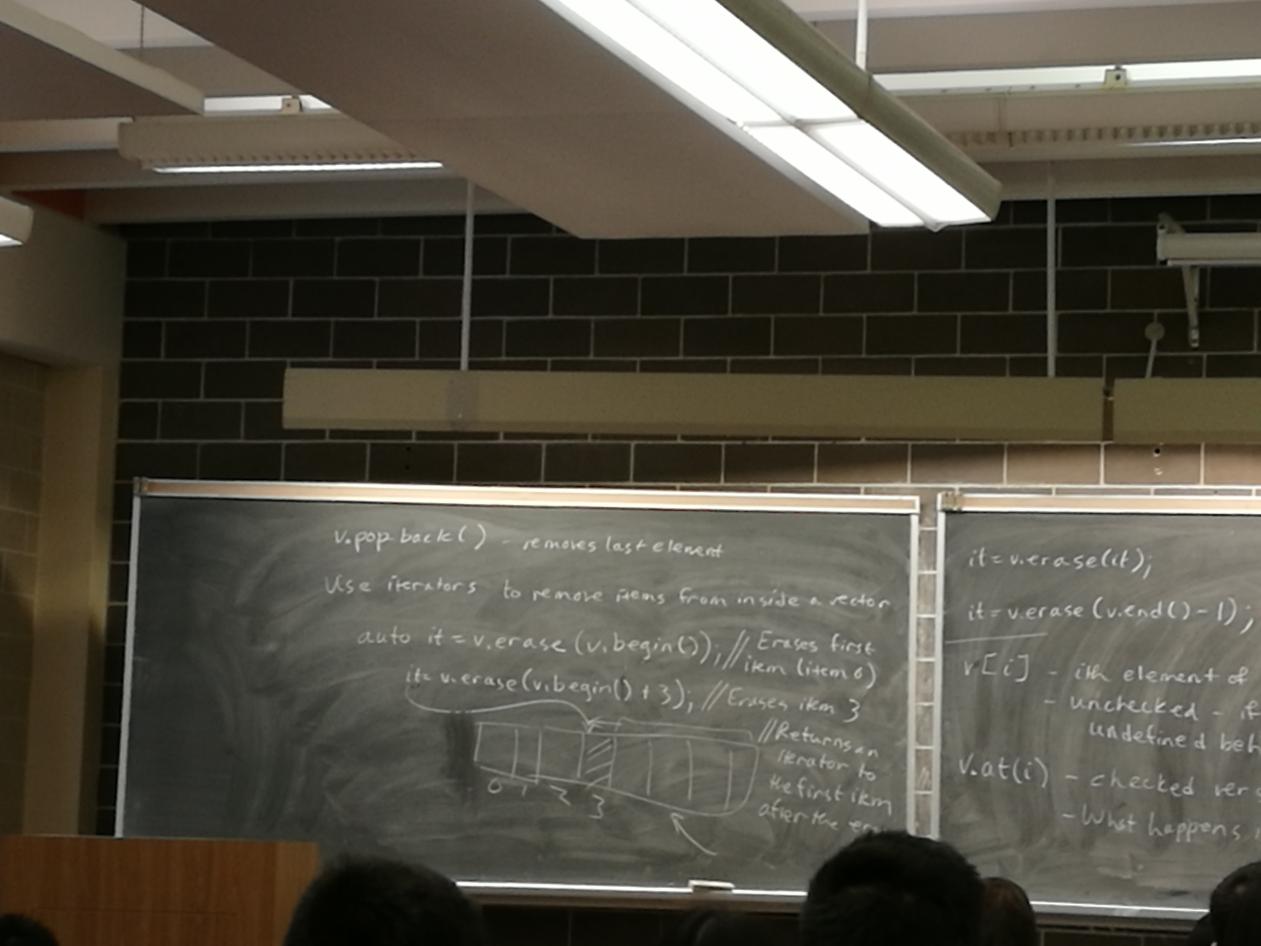
}

v.pop\_back() -removes last element

use iterators to remove items from inside a vector

auto it = v.erase(v.begin()); // Erases first item (item 6)

it = v.erase(v.begin() + 3);// Erase item 3

[]

it = v.erase(it);

it = v.erase(v.end() - 1); //last item

v[i] - ith element of v

- unchecked - if you go out of bounds,undefined behavior

v.at(i) - checked version of v[i]

- What happens if you go out of bounds?

What should happen?

Problem : vector’s code can detect the error, but doesn’t know what to do about it

-client can respond, but can’t detect the error

C solution: functions return a status code, or set the global variable errno

-leads to awkward programming

- encourages programmers to ignore error checks

C++: When an error condition arises, the function raises an exception

What happens? - By default execution steps.

But we can write handlers to catch exns of deal with them.

Vector<T>:: at raises the exn std:: out\_of\_range

we handle as follows:

#inlcude <stdexcept>

...

try { // stmts that my fail go into a try block

cout << v.at(10000) <<endl;

}

catch (out\_of\_range) {

cerr<< “Range error” << endl;

}

11.3

Now Consider ctor call

void f() { throw out of range {“f”};}

create an object what()

of type & throw it out\_of\_range

void g() {f();}

void h() {g();}

int main() {

try {

h();

}

catch (out\_of\_range) {...}

}

What happens:

main calls h

h calls g

g calls f

f throws

-g has no handler for out\_of\_range

-Control goes back through the call chain(unwinds the stack) until a handler is found

-Control goes all the way to main,which handles the exn

If no handler in the entire call chain, program terminates

What is out\_of\_range?

-A class throw out\_of\_range {“f”} -ctor call

A handler could do part of the recovery job

-execute some corrective code & throw another exn

try{...}

catch (someErrorType s) {

...

throw SomeOtherError {....};

}

-or throw the same exn

try {...}

catch (someErrorType s) {

...

throw;

}

throw vs. throw s SomeError Type

actual type ↑

of S is retained SpecialError Type

throws a new exn(exception) of type SomeError Type

s may have originated as a subclass of SomeError Type

A handler can act as a catch-all

try{---}

catch(...) { // catches all exns //not ellipsis but real dot dot dot

-------

}

You can throw anything you want

-don’t have to throw objects

Define your own exn classes

class BadInput{};

try {

int n;

if (!(cin >> n)) throw BadInput{};

}

catch(BadInput &) {

cerr << “------” << endl;

}

Exception std::bad\_alloc -thrown when new fails

---------------------------------------------------------------------------

Design Patterns ctd

Guiding principle:program to the interface, not the implementation

-abstract base classes define the interface

-work with ptrs to abstract base classes, and call their methods

-concrete subclasses can be swapped in + out

-abstract over a variety of behaviors

Eg Iterator pattern:

class List {

...

public:

class Iterator:public AbstractIterator {

...

};

...

};

Class AbstractIterator{

public:

virtual int &operator \*()=0;

virtual AbstractIterator &operator ++() = 0;

virtual bool operator != (const AbstractIterator & other) = 0;

virtual ~AbstractIterator();

};

class Set {

...

public:

class Iterator:public AbstractIterator {

...

};

}; Then you can write code that operates over iterators

void foreach (AbstractIterator &start, AbstractIterator &end, void(\*f)(int)) {

while (start != end) {

f(\*start);

++start;

}

} -works over List and Sets

Observer Pattern

Publish-subscribe model

One class: publisher/subject - generates data

One or more subscribe/observer classes

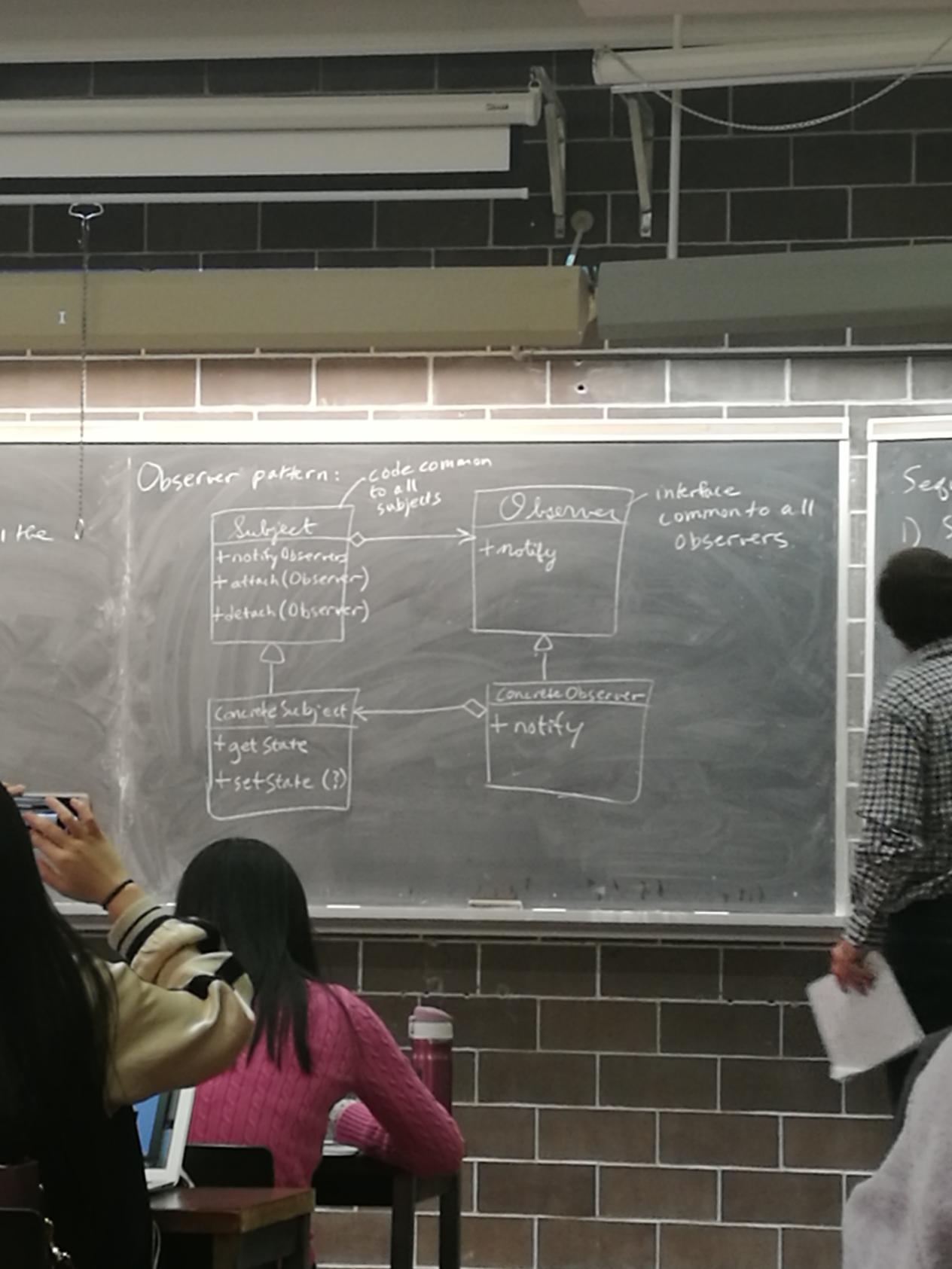
-receive data + react to it

Eg publisher-spreadsheet-cells

observers - graphs

can be many different kinds of observers

-subject should not need to know all the details

[]

Sequence of method calls:

1. Subject’s state is updated
2. Subject::notifyObservers() - calls each observer’s notify
3. Each observer calls ConcreteSubject:: get state to query the state of act accordingly

Example: Horse races

- Subject: publishes winners

- Observers : individual bettors

- declare victory when their horse wins

class Subject{

vector<Observer\*>observers;

public:

bool attach(Observer\*o) {observers.emplace-back(o);}

void detach(Observer\*o); // exercise

void notifyObservers() {

for (auto &ob:observers) ob->notify();

}

virtual ~Subject()=0;

};

Subject::~Subject(){} // virtual dtor must always be implemented, even if it is pure virtual

Object destruction:

1)dtor body

2)fields destructed

3)superclass dtor \* so there had better be one

4)space deallocated

class HorseRace :public Subject {

ifstream in; //source of data

string lastWinner;

public:

HorseRace(string source) : in {source}{}

~HorseRace();

bool runRace(); // sets lastWinner - true if successful

string getState() { return lastWinner;}

};

11.8

class Observer {

virtual void notify()=0;

virtual ~Observer(){}

};

class Bettor:public Observer {

HorseRace \*subject;

string name,myHorse;

public:

Bettor(----):------{

subject->attach(this);

}

~Bettor() {subject->detach(this);}

void notify() override {

string winner = subject->getState();

if (winner == myHorse) {

cout “Yay!”;

} else cout << “Double or nothing”;

}

};

main: HorseRace hr {-------};

Bettor Larry(&hr, “Larry”,”RunsLikeACow”);

...

While (hr.runRace()) {

hr.notifyObservers();

}

Decorator Pattern

Want to enhance an object at run-time

-add functionality/features

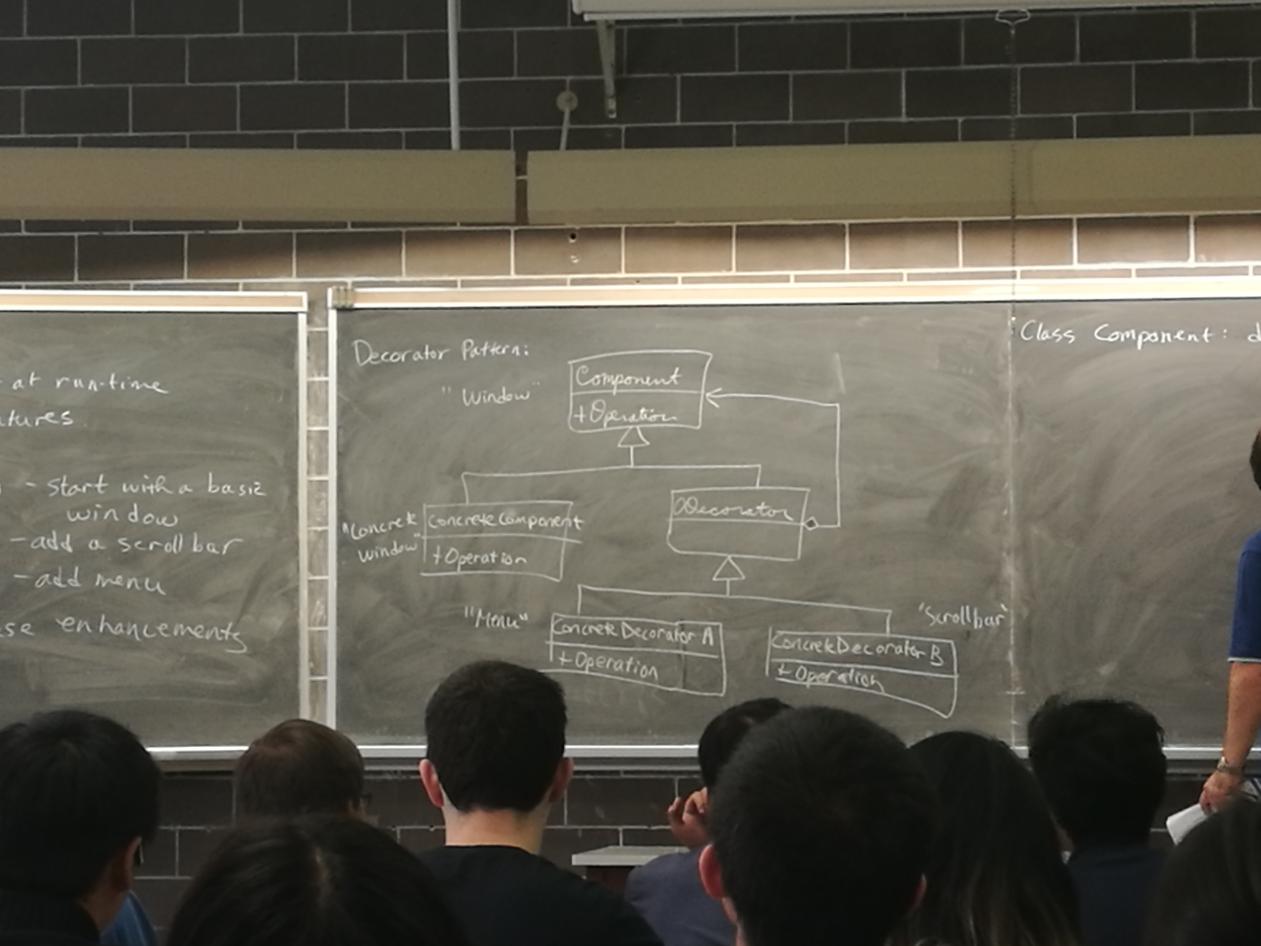
Eg: Windowing system - start with a basis window

-add a scrollbar

-add menu

Want to choose these enhancements at run-time

Decorator Patter:

[]

class Component: defines the interface - operations

your objects will provide

Concrete component -implement the interface

Decorators-all inherit from Decorator

which inherit from Component

so Every decorator is a Component AND every decorator has a Component

Eg Window with scrollbar is a kind of window, and has a ptr to the underlying plain window

Window w/ scrollbar & menu is a window, has a ptr to a window w/ scrollbar, which has a ptr to a plain window

-All inherit from Abstract Window class, so Window methods can be used polymorphically on all of them

Eg: PIZZA

[]

Basic Pizza is crust and sauce

class Pizza {

public :

virtual float price() const = 0;

virtual string desc() const = 0;

virtual ~Pizza();

};

class crustAndSauce:public Pizza{

public:

float price() const override {return 5.99;}

string desc() const override {return “Pizza”;}

};

class Decorator:public Pizza {

protected:

Pizza \*component;

public:

Decoretor(Pizza \*p): component {p} {}

virtual ~Decorator() {delete component;}

}; ?

class Topping : public Decorator {

string theTopping;

public:

Topping (Pizza \*p, string topping):Decorator{p},theTopping{topping} {}

float price() const override {return component->price()+.75;}

string desc() const override {

return component->desc + “with” + theTopping;

}

};

use:

Pizza \*p1 = new CrustAndSauce;

p1 = new Topping(p1,”Cheese”)

p1 = new Topping(p1,”Jelly Beans”);

p1 = new stuffedCrust(p1);

cout << p1->dest() << ‘ ‘ << p1->price();

delete p1;

Inheritance and copy/move

class Book {

// defines copy/move ctors, copy/move assign

};

class Text:public Book{

// Does not define copy/move operations

};

Text t{“Algorithms”,”CLRS”,500,”CS”};

Text t2 = t; //No copy ctor in Text - What happens?

-calls Book’s copy ctor

-then goes field-by-field (i.e default behavior) for the Text part

-same for other operation

To write your own operations,

Text:: Text(const Text &other):Book{other},topic{other.topic}{}

Text & Text::operator=(const Text&other) {

Book::operator= (other);

topic = other.topic;

return \*this;

}

Text::Text(Text &&other):

Book{std::move(other)}, topic{std::move(other.topic)} {}

Text &Text::operator= (Text &&other) {

Book::operator=(std::move(other));

topic=std::move(other.topic);

return \*this;

}

Note: even though other “points” at an rvalue, other itself is an Ivalue(so is other.topic)

std::move(x) forces an Ivalue x to be treated as an rvalue so that the “move” version of the operators run.

Operations given above are equivalent to the compiler-supplied defaults

Now consider:

Text t1 {--------},t2{--------};

Book \*pb1=&t1, \*pb2 =&t2;

What if we do:\*pb1 = \*pb2;

Partial assignment - copies only the book part

-Because Book:operator= runs.

How can we fix this? Try making operator= virtual

class Book {

...

public:

virtual Book &operator= (const Book &other);

};

class Text:public Book {

...

public: Book

Text &operator =(const Text &other) override;

};

Note: different return types OK (as long as you return a subclass by ref/ptr),but the parameter types must be the same, or it’s not an override, won’t compile, and violates is-a.

therefore, Assignment of a Book Obj. to a Text var would be allowed:

Text t{...};

Book b{...};

Text \*pt=&t;

Book \*pb=&b;

\*pt=\*pb; //- Uses a Book to assign a Text

- BAD -but it compiles

Also Comic c{...}

Comic \*pc = &c;

\*pt = \*pc; // true,but really BAD

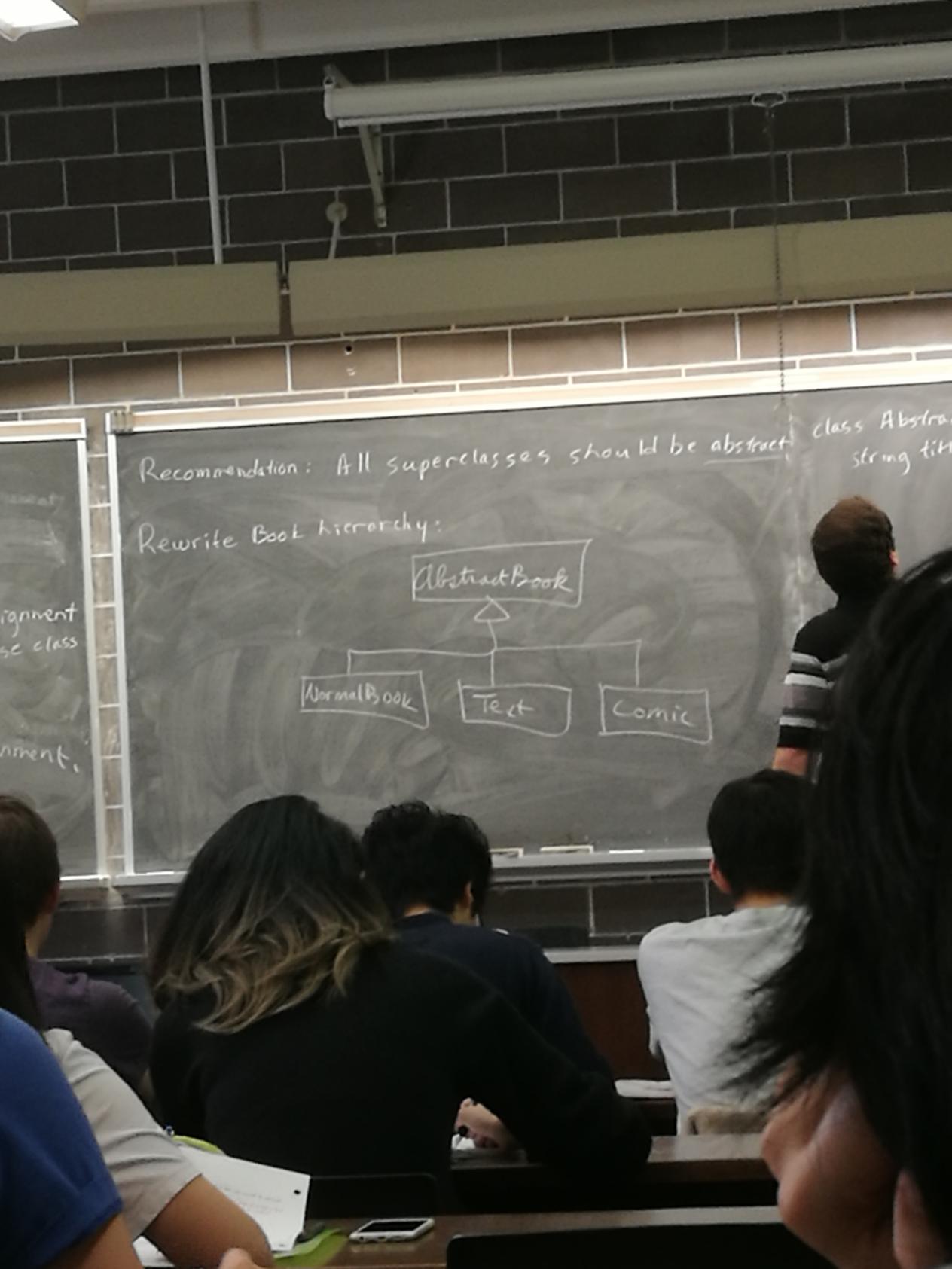
----------------------------------------------------------------------------

If operator= not virtual -partial assignment through base class ptrs

If virtual compiler allows mixed assignment

Recommendation: All superclasses should be abstract

Rewrite Book hierarchy:

[]

class AbstractBook {

string title,author;

int numPages; prevents assignment through base class ptrs from compiling,but

implementation still available on subclasses

protected: |

AbstractBook &operator=(const AbstractBook&other);

public:

AbstractBook(---);

virtual ~AbstractBook()=0;

};

class NormalBook:public AbstractBook{

public:

NormalBook(--);

~NormalBook();

NormalBook &operator=(const NormalBook &other) {

AbstractBook::operator=(other);

return \*this;

}

}; // other classes-exercise

Text t1 {...}, t2{...};

Book \*pb1=&t1, \*pb2=&t2;

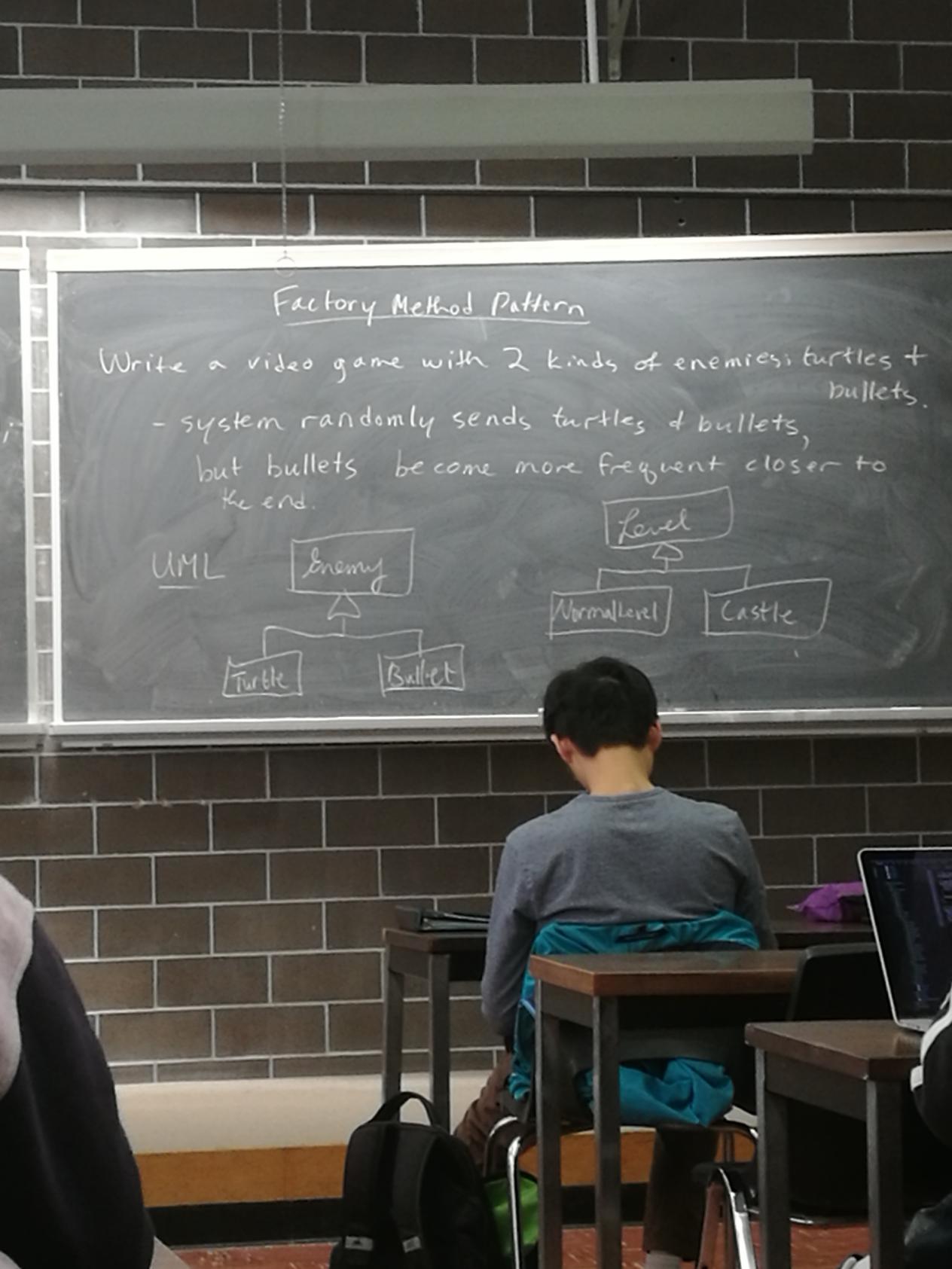
\*pb1=\*pb2; //X(false) doesn’t compile

Factory Method Pattern

Write a video game with 2 kinds of enemies: turtles + bullets

-system randomly sends turtles + bullets,

but bullet become more frequent closer to the end

[]

Never know exactly which enemy comes next, so can’t call ctors directly

Put a factory method in Level that creates enemies

class Level{

public:

virtual Enemy\*createEnemy()=0;

};

class NormalLevel:public Level {

public:

Enemy\*createEnemy() override{

//create mostly turtles

}

};

class Castle:public Level {

public:

Enemy \*createEnemy() override {

//mostly bullets

}

};

Level \*l=new NormalLevel;

Enemy\*e = l->createEnemy();

Template Method Pattern

Want subclasses to override superclass behavior,but some aspects must stay the same

Eg- there are red turtles + green turtles

class Turtle {

public: //template method

void draw() {

drawHead();

drawShell();

drawTail();

}

private:

void drawHead();

void drawFeet();

virtual void drawShell()=0;

};

class RedTurtle:public Turtle {

void drawShell() {/\* draw redShell\*/}

};

class GreenTurtle:public Turtle{

void drawShell() {/\* draw green shell \*/}

};

subclasses can’t change the way a turtle is drawn(i.e head,then shell,then feet),but can change the way the shell is drawn.

Extension - the Non-Virtual Interface (NVI) idiom

- A public virtual method is really two things:

- an interface to the client

- indicates provided behavior with pre/post conditions

- an interface to subclasses

- a “hook” to insert specialized behavior

Hard to separate these ideas if they are tied to the same func.

-What if you want to separate the customizable behavior into two fns, with some unchanging code in between, while still providing clients the same interface?

- How could you make sure that overriding fns conform to pre/post conditions?

NVI says :- all public methods should be non-virtual

- all virtual methods should be private, or at least protected

- except the dtor

Example:

class DigitalMedia {

public:

virtual void play()=0;

};

~~~~~~>

class DigitalMedia {

public; can add before/after code e.g check copyright

void play() { ↑doPlay();↓}

private: update playcount

virtual void doPlay()=0;

};

Extends template method - every virtual method is inside a template method

STL Maps-for creating Dictionaries

eg “arrays” that map strings to ints

#include <map>

map<string,int> m;

m[“abc”] = 1;

m[“def”] = 2;

cout << m[“ghi”];// if key not present, it is inserted, & value is default -constructed(for ints,0)

cout << m[“abc”]; // 1

m.erase(“abc”);

if (m.cout(“def”)) ... 1 = found

0 = not found

Iterating over a map: sorted key order

for (auto &p:m) {

cout << p.first << ‘ ‘ << p.second<<endl;

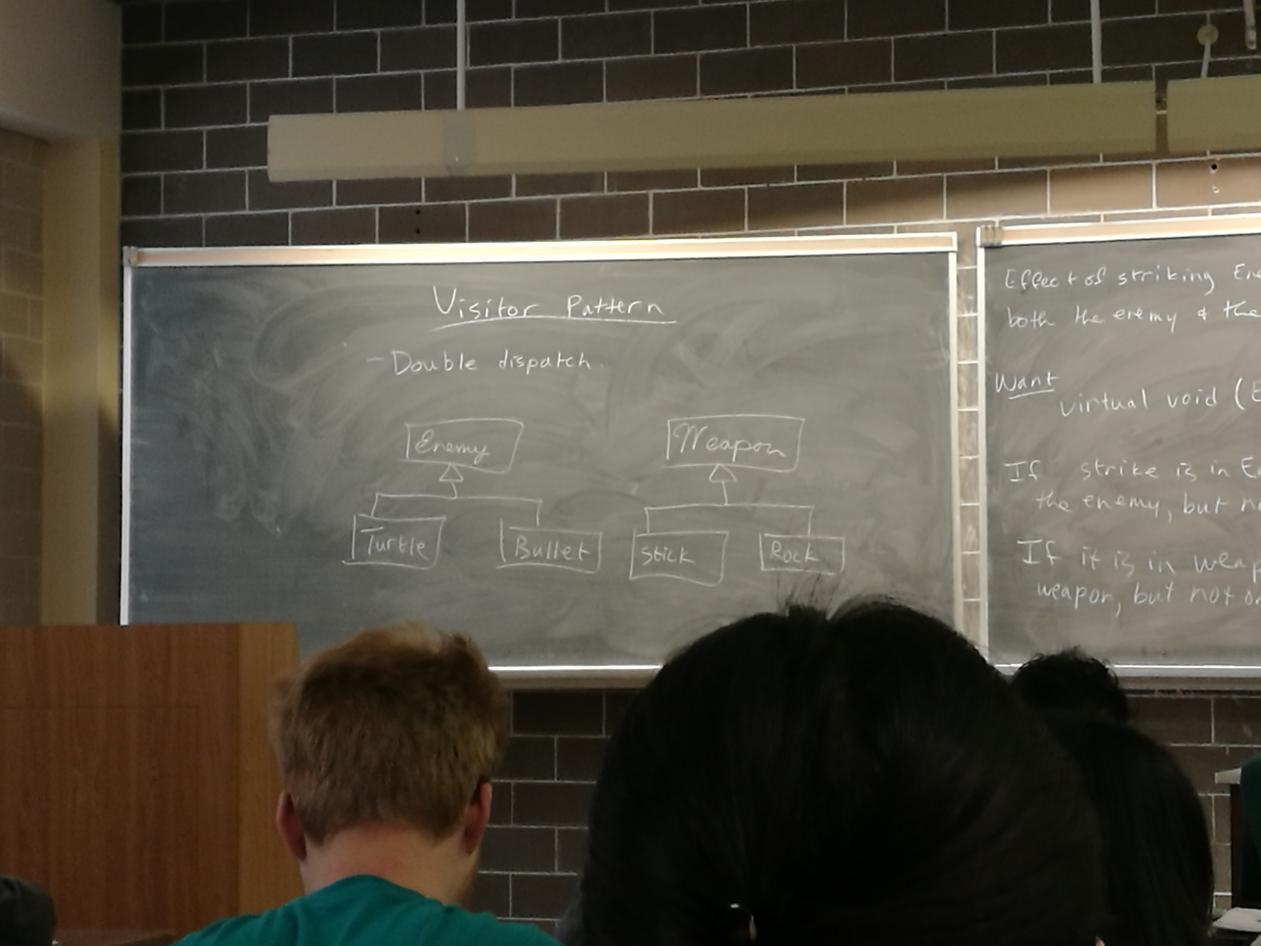
}

p’s type is std::pair<string,int>& (<utility>)

11.15

Visitor Pattern

-Double dispatch

[]

Effect of striking Enemy with weapon depends on both the enemy & the weapon

Want

virtual void (Enemy,Weapon)::strike()

If strike is in Enemy,effect will depend on the enemy,but not the weapon.

If it is in weapon, effect will depend on the weapon, but not on the enemy

Trick to get dispatch based on both(double dispatch)-combine overriding with overloading

class Enemy {

virtual void beStruckBy(Weapon &w) = 0;

};

class Turtle:public Enemy {

void beStruckBy(Weapon &w) override { w.strike(\*this);}

};

class Bullet:public Enemy {

void beStruckBy(Weapon &w) override { w.strike(\*this);}

};

class Weapon {

virtual void strike(Turtle& t)=0;

virtual void strike(Bullet &b) = 0; //these two overload

};

class Stick:public Weapon {

void strike (Turtle &t) override {

//strike turtle with stick

}

void strike(Bullet &b)override {

// bullet with stick

}

};

Rock-similar

Enemy \*e = new Bullet{...};

Weapon \*w = new Rock {...};

e->beStruckBy(\*w); What happens!

virtual

Bullet::beStruckBy- calls Weapon::strike,\*this is a Bullet

So this resolves,at compile-time, to Weapon::strike(Bullet &)

virtual

-resolves to Rock::strike(Bullet &)

Visitor can be used to add functionality to existing classes,without changing or recompiling them.

Eg add a visitor to the Book hierarchy

class Book {

public:

virtual void accept(BookVisitor &v){

v.visit(\*this);

}

};

class Text:public Book{

public:

void accept(BookVisitor &v) override {

v.visit(\*this);

}

};

etc.

class BookVisitor{

public:

virtual void visit(Book &b)=0;

virtual void visit(Text &t) = 0;

virtual void visit (Comic &c) = 0;

};

Application:Track how many of each type of Book I have

Books - byAuthor

Texts - by topic

Comics - by hero

Use a map<string,int>

Could add virtual void updateMap() {...}

Or write a visitor:

class Catalogue : public BookVisitor{

map<string,int> theCatalogue;

public:

map<string,int> getResult() { return theCatalogue;}

void visit(Book &b) override{

++theCatalogue[b.getAuthor()];

}

Doesn’t compile ! Why?

main.cc includes book.h,includes Bookvisitor.h,includes Text.h,includes book.h X (include guard)

-Are these includes really needed?

Compilation Dependencies-include vs. forward declare

Consider: class A {...}

#include “a.h”

Class B:public A {

...

};

#include “a.h”

class C {

A myA;

}; // B,C:Need to know A’s size to construct a B or C object

class A;

class D {

A \*myA;

};

class A;

class E{

A f (A x);

};

Don’t introduce extra compilation dependencies by unneeded #includes

Now, in the implementations of D,E:

d.cc:

void D::f() {

myAp->someMethod(); - need to know A’s methods

} real compilation dependency

Do the #include in the .cc,instead of the .h(where possible)

Now consider the XWindow class:

class XWindow {

Display \*d;

Window w;

int s;

Gc gc;

unsigned long colours[10]; //This is private data

public: Yet we can look at it. Do we know what it all

... means? Do we care?

};

If we add or change a private member - all clients must recompile. Would be better to hide these details away.

Sol’n: pimpl idiom (pointer to implimentation)

Create a second class XWindowImp:

XWindowImpl.h:

#include <XII/Xlib.h>

struct XWindowImpl{

Display \*d;

Window w;

int s;

Gc gc;

unsigned long colors[10];

};

Window.h

// no need to include Xlib.h

//forward declare the Impl class

class XWindowImpl;

class XWindow {

XWindowImpl \*pImpl; |

public: |

... //no change |No compolation dependency on XWindowImpl.h

}; Clients also don’t depend on XWindowImpl.h

window.cc

#include “window.h”

#include “XWindowImpl.h”

XWindow::XWindow(...):

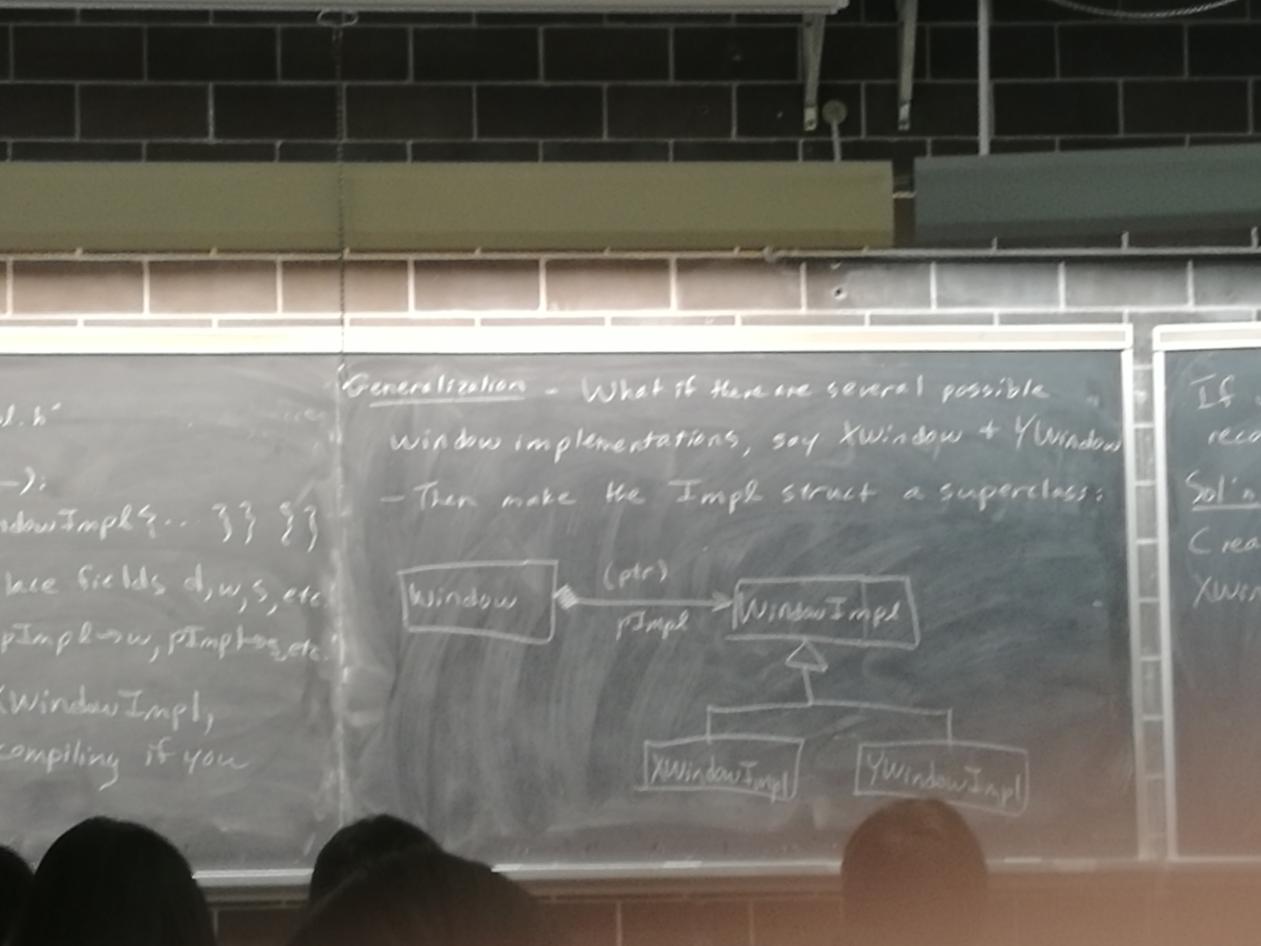
pImpl {new XWindowImpl{..}}{}

Other methods:replace fields d,w,s etc with pImpl->d,pImpl->w,pImpl->s etc

If all provide fields are in XWindowImpl, then only window.cc needs recompiling if you XWindows implementation.

Generalization - What if there are several possible window implementation, say XWindow + YWindow

- Then make the Impl struct a superclass:

[]

11.17

Measures of Design Quality

coupling and cohesion

coupling:the degree to which distinct program modules depend on each other

low:-modules communicate via function calls w/ basic params/results

- modules pass arrays/structs back and forth

- modules affect each other’s control flow

- modules share global data

high: -module have access to each other’s implementation(friends)

cohesion: - how closely elements of a module are related to each other

low: - arbitrary grouping of unrelated elements (e.g.<utility>)

- elements share a common theme, but otherwise unrelated - perhaps share some base code (eg <algorithm>)

-elements manipulate state over the lifetie of an object(eg. open/read/close files)

-elements pass data to each other

high - elements cooperate to perform exactly one task

----------------------------------------------------------------------------------------

high coupling => changes to one module greater changes to other modules

- harder to reuse individual modules

low cohesion => poorly organized code

- hard understand, hard to maintain

Goal: low coupling,high cohesion

Decoupling the Interface (MVC)

Your primary program classes should not be printing things

Eg

class ChessBoard {

....

cout << “Your move” << endl;

};

Bad design - inhibits code reuse

What if you want to reuse ChessBoard, but not have it communicate via stdout?

One solution: give the class stream objects to read from lwrite to

class ChessBoard {

istream &in;

ostream &out;

public :

ChessBoard (istream &in,ostream &out):

in{in},out {out} {...}

...

out << “Your move” << endl;

};

Better - but what if we don’t want to use streams at all?

You chessboard class should not be doing any communication at all;

Single Responsibility Principle: “A class should have only one reason to change”

game state + communication are two reasons.

Better: Communication withe the Chessboard via params/results

-Confine user communication to outside the game class

Q: Should main do all of the communication & then call chessboard methods?

A: No - Hard to reuse code if it’s in main

Should have a class to manage interaction, that is separate from the game state class

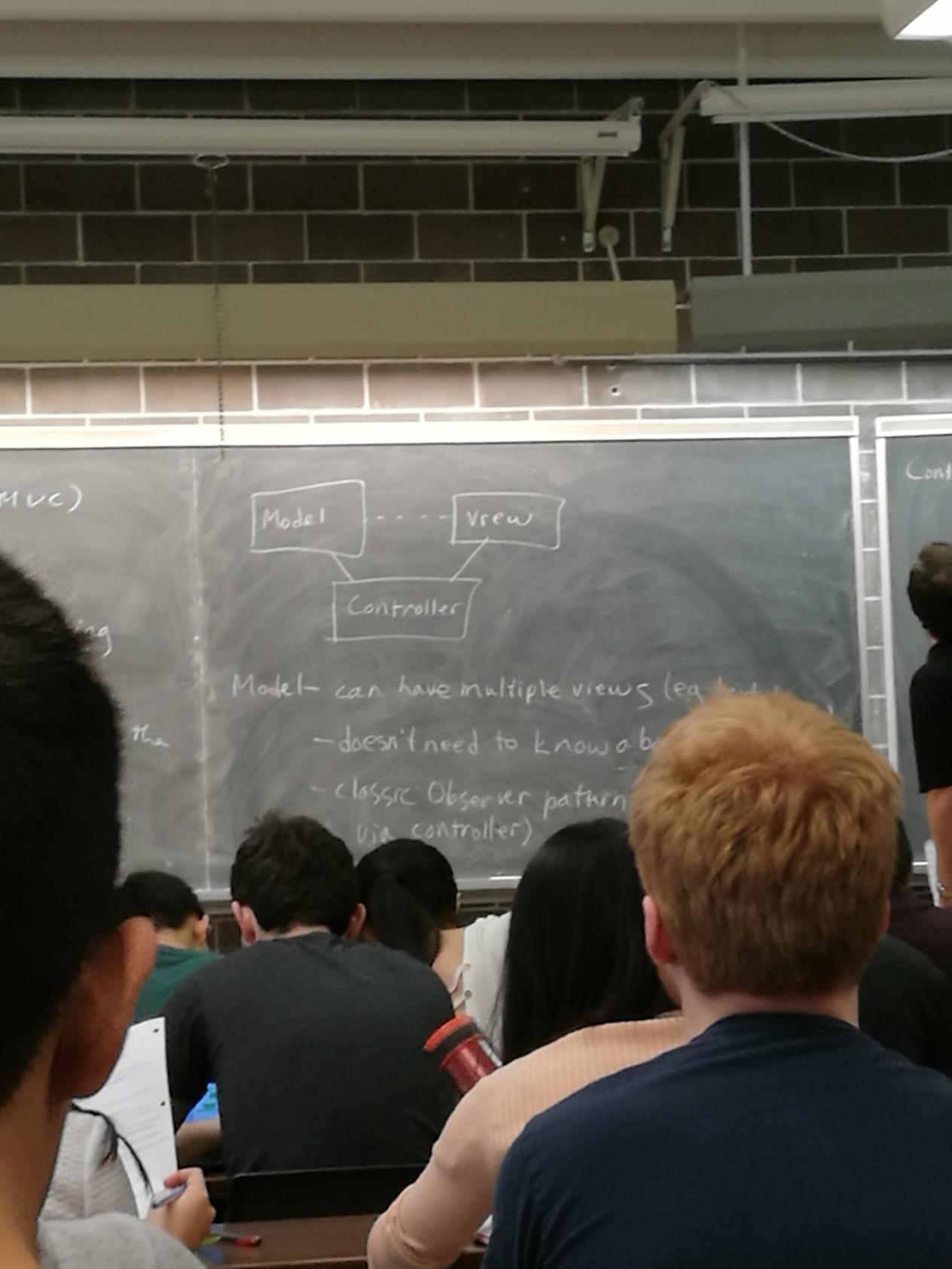
Pattern: Model - View- Controller (MVC)

Separate the distinct notions of

Model - the data you are manipulating (eg game state)

View - How the model is displayed to the user

Controller - How the model is manipulated



Model - can have multiple viewS (eg text + graphics)

-doesn’t need to know about their details

- classic Observer pattern (or could communicate via controllers)

Controller - mediates control flow through model &view

- may encapsulate turn- taking, or full game rules

- may fetch user input (or this could be the view)

-By decoupling presentation & control, MVC promotes reuse

Exception safely

Consider:

void f() {

MyClass \*p = new MyClass;

Myclass mc;

g();

delete p;

}

No leaks - what if g raises an exn?

What is guaranteed?

- During stack-unwinding, all stack-allocated data is cleaned up - dtors run,

memory is reclaimed

-so mc is reclaimed

-Heap-allocated memory is not deleted

so If g throws, p is leaked;

void f() {

MyClass \*p = new MyClass;

MyClass mc;

try {

g();

}

catch(...){

delete p; |

throw; |

} |

delete p; |

} | // Tedious & error prone - code duplication

How else can we guarantee that something (eg delete p)

will happen no matter how we exit f (normally or by exn)?

In some languages - “finally” clauses guarantee certain final actions - not in C++

Only thing you can count on in C++ - dtors for stack-allocated data will run.

Use stack- allocated data with dtors as much as possible

- use this guarantee to your advantage

C++ idiom:RAII - Resource Acquisition Is Initialization

Every resource - should be wrapped in a stack-allocated object, whose dtor frees it.

Eg files { ifstream f {“name”};

Acquiring the resource (“name”) = initializing the object (f)

-the fine is guaranteed to be released when f is popped from the stack (f’s dtor runs)

}

This can be done why dynamic memory:

class std::unique-ptr<T> - Takes a T\* in the ctor

- the dtor will delete the ptr;

in between - can dereference, just like a ptr

void f() {

auto p = std::make\_unique<MyClass>();

--ctor args

//type is std::unique-ptr<MyClass>;

MyClass me;

g();

} - No leaks Guaranteed

Difficulty:

class c {...};

unique\_ptr<C> p {new c{}};

unique\_ptr<C>q = p;

What happens when a unique\_ptr is copied? - Don’t want to delete the same ptr twice!

Instead-copying is disabled for unique ptrs

They can only be moved

11.22

Sample implementation:

template <typename T> class unique\_ptr {

T \*ptr;

public:

unique\_ptr(T \*p):ptr{p}{}

~unique\_ptr() {delete ptr;}

unique\_ptr(const unique\_ptr<T> &other) = delete;

unique\_ptr<T> &operator=(const unique\_ptr) &other) =delete;

unique\_ptr(unique\_ptr<T> &&other) : ptr{other.ptr} {other.ptr = nullptr;}

unique\_ptr<T> &operator=(unique\_ptr<T> &&other) {

using std::swap;

swap(ptr,other.ptr);

return \*this;

}

T &operator\*() { return \*ptr; }

};

If you need to be able to copy ptrs:

std::shared\_ptr

{ auto p1 = std::make\_shared<Myclass>();

if (...) {

auto p2 = p1;

} // p2 popped,ptr is not deleted

...

} // p1 popped,ptr is deleted

Shared ptrs maintain a reference count - count of all shared-ptrs pointing at the same object

Memary is feed when the count hits 0.

Use shared-ptr & unique-ptr instead of raw ptrs, whenever ownership is intended

Dramatically fewer opportunities for leaks

NEVER let a dtor emit an exn

-if the dtor was executed during stack unwinding , while dealing with another exn, you now have two active, unhandled exns, and the program will abort immediately

3 levels of exception safety for a f’n f:

1. Basic guarantee - if an exn occurs, the program will be in a valid state.

-no leaks, no corruption, class invariants maintained

1. Strong guarantee - if an exn is raised while executing f, the state of the program will be as if f had not been called
2. No-throw guarantee - f will never throw and exn, and will always accomplish its task

Example:

class A {..};

class B {...};

class C {

A a;

B b;

void f () {

a.method1();//

b.method2();//May throw(strong guarantee)

}

};

Is C::f exn safe?

- if method1 throws, nothing has happened yet - OK

- if method2 throws, effects of method1 must be undone to after the strong guarantee

-very hard or impossible if method1 has non-local side-effects

so No,probably not exn safe

If method1 & method2 do not have non-local side-effects,can use copy+swap

class C {

...

void f() {

A atemp = a;

B btemp = b;

atemp.method1();

btemp.method2();// if these throw, f still throws but original a+b intact

a = atemp;

b = btemp;//What if these throws?

}

};

Better if the swap was nothrow - copying ptrs cannot throw

Sol’n - use the pImpl idiom:

struct CImpl {

A a;

B b;

};

class C {

unique\_ptr<CImpl> pImpl;

...

void f() {

auto temp = make\_unique<CImpl>(\*pImpl);

temp->a.method1(); strong guarantee

temp->b.method2();

std::swap(pImpl,temp); // No-throw

}

};

If either method1 or method2 isn’t exn safe,then neither is f

Exception Safety & the STL

vector - encapsulates a heap-allocated array:

-RAII - when a stack-allocated vector goes out of scopes, the internal heap array is freed

void f() {

vector<Myclass> v;

...

} //v goes out of scope - array is freed, Myclass dtor runs in all objs in the vector

But

void g() {

vector <Myclass\*> v;

...

}// array is freed, ptrs don’t have dtors, so any objects pointed to by the ptrs are not deleted

-v doesn’t know whether the ptrs in the array own the objs they point at

if they do: for (auto x:v) {delete x;}

But:

void h() {

vector<shared-ptr<Myclass>> v;

...

} // array is freed, share\_ptr dtors run, so objes are deleted if no other shared ptr points at them.

-Don’t have to do any explicit deallocation.

11.24

Exception safety + the STL continued

vector<T>::emplace-back -provides the strong guarantee

- if the array is full (i.e. size-cap)

-allocate a new array

-copy objs over (copy ctor) - if a copy ctor throws

-destroy the new array

-old array still intact

-strong guarantee

-delete old array

But

-copying is expensive,

&the old data will be thrown away

-wouldn’t moving the objs be more efficient?

-allocate new array

-move the objs over(move ctor) - if move ctor throws, can’t offer the strong guarantee

-original no longer intact

-delete old array

If the move ctor offers the no throw guarantee,emplace-back will use the move ctor;else it will use the copy ctor, which will be slower

So your move ops should provide the no-throw guarantee, and you should indicate that they do:

class MyClass {

public:

MyClass(MyClass &&other) noexcept {...}

MyClass &operator= (MyClass &&other) noexcept;

};

If you know a f’n will never throw or propagate an exn, declare it noexcept - facilitates optimization

At minimum: swaps & moves should be noexcept

-------------------------------------------------------------------------------------------------------------------------------

Casting

In C:

Node n;

int \*ip = (int\*)(&n),

-cast - forces C++ to treat a Node\* as an int\*

C-style casts should be avoided in C++. If you must cast, use a C++ -style cast

4 kinds:

- static\_cast - “sensible” casts

Eg double->int

double d;

void f(int i); // 1

void f(double d);

f(static\_cast<int>(d)); //call 1

superclass ptr to subclass ptr

Book \*b = new Text{..};

Text \*t = static\_cast<Text \*>(b);

-You are taking responsibility that b actually points at a Text. “Trust me”

-reinterpret\_cast - Unsafe,implementation-specifie, “Weired casts”

Student s;

Turtle \* t = reinterpret\_cast<Turtle \*>(&s);

- const\_cast - for converting between const & non-const

- the only C++ cast that can “cast away const”

void g(int \*p); //Given to you

void f(const int \*p) {

...

g(const\_cast<int \*>(p));

...

} // suppose you know that g doesn’t actually notify \*p - just not indicated in the

header

dynamic\_cast -Is it safe to convert a Book \* to a Text \*?

Book \*pb;

...

static\_cast<Text \*>(pb)->getTopic();

safe?

-Depends on what pb actually ptr at.

-Better to do a tentative cast - try it and see if it succeeds

Book \*pb = ...;

Text \*pt = dynamic\_cast<Text\*>(pb);

If the cast works(pb really points at a Text, or a subclass of Text), pt points at the object.

If the cast fails, pt will be nullptr.

if (pt) cout << pt->getTopic(),

else cout << “Not a Text”;

Should be using smart ptrs - can we do this? Yes

static\_pointer\_cast

const\_pointer\_cast

dynamic\_pointer\_cast //all of above, cast shared\_ptrs to shared\_ptrs

Can use dynamic casting to make decisions based on an object’s RTTI (run-time type intormation)

void whatIsIt(shared\_ptr<Book> b) {

if (dynamic\_pointer\_cast <Comic>(b))

cout << “Comic”;

else if (dynamic\_pointer\_cast<Text>(b))

cout << “Text”;

else cout << “Normal Book”;

}

Code like this is tightly coupled to the Book hierarchy and may indicate bad design.

Better:-use virtual methods

- write a visitor

---------------------

Dynamic casting also works with references:

Text t {...};

Book &b = t;

Text &tz = dynamic\_cast<Text&>(b);

If b “points to” a Text, tz is a ref to the same Text.

If not ...? (No such thing as a null reference)

-raises exn bad\_cast

Note: dynamic casting only works on classes that have at least one virtual method

11.29

Can use this to solve the polymorphic assignment problem:

Text &Text::operator=(const Book &b) { // virtual

Text &textother= dynamic\_cast<Text &>(b);

-throws an exn if other is not a Text

if (this == &textother) return \*this;

Book::operator = (other);

topic = textother.topic;

return \*this;

}

How Virtual Method Work

class Vec {

int x,y;

public:

void doSomething();

};

class Vec2 {

int x,y;

public:

virtual void doSomething();

};

What’s the differences?

Vec v{1,2};

Vec2 w{1,2}; // Do they look the same in memory?

cout << sizeof(v) << ‘ ‘ << sizeof(w);

8 16 ????

First note: 8 = space for 2 ints.

- No space for the doSomething method

Compiler turns methods into ordinary f’ns + stores them separately from obj’s

Recall: Book \*pb = new (Book,Text,Comic)

pb->isItHeavy();

isItHeavy is virtual - choice of which version of

isItHeavy to run is based on the type of the actual object - which the compiler can’t know in advance.

so, Correct isItHeavy must be chosen at run-time, how?

For each class with virtual methods the compiler creates a table of f’n ptrs(the vtable)

Eg

class c {

int x,y;

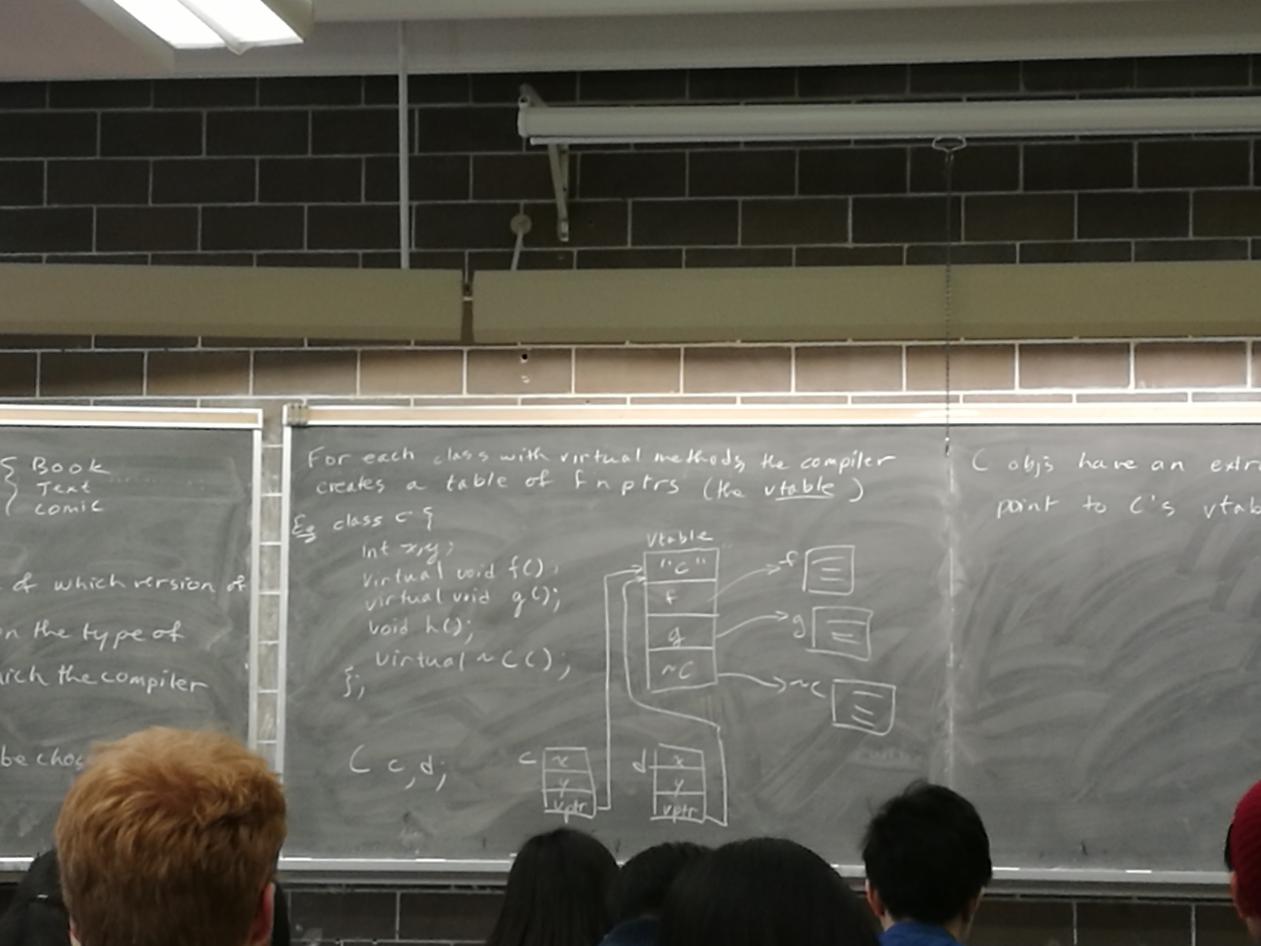
virtual void f();

virtual void g();

void h();

virtual ~c();

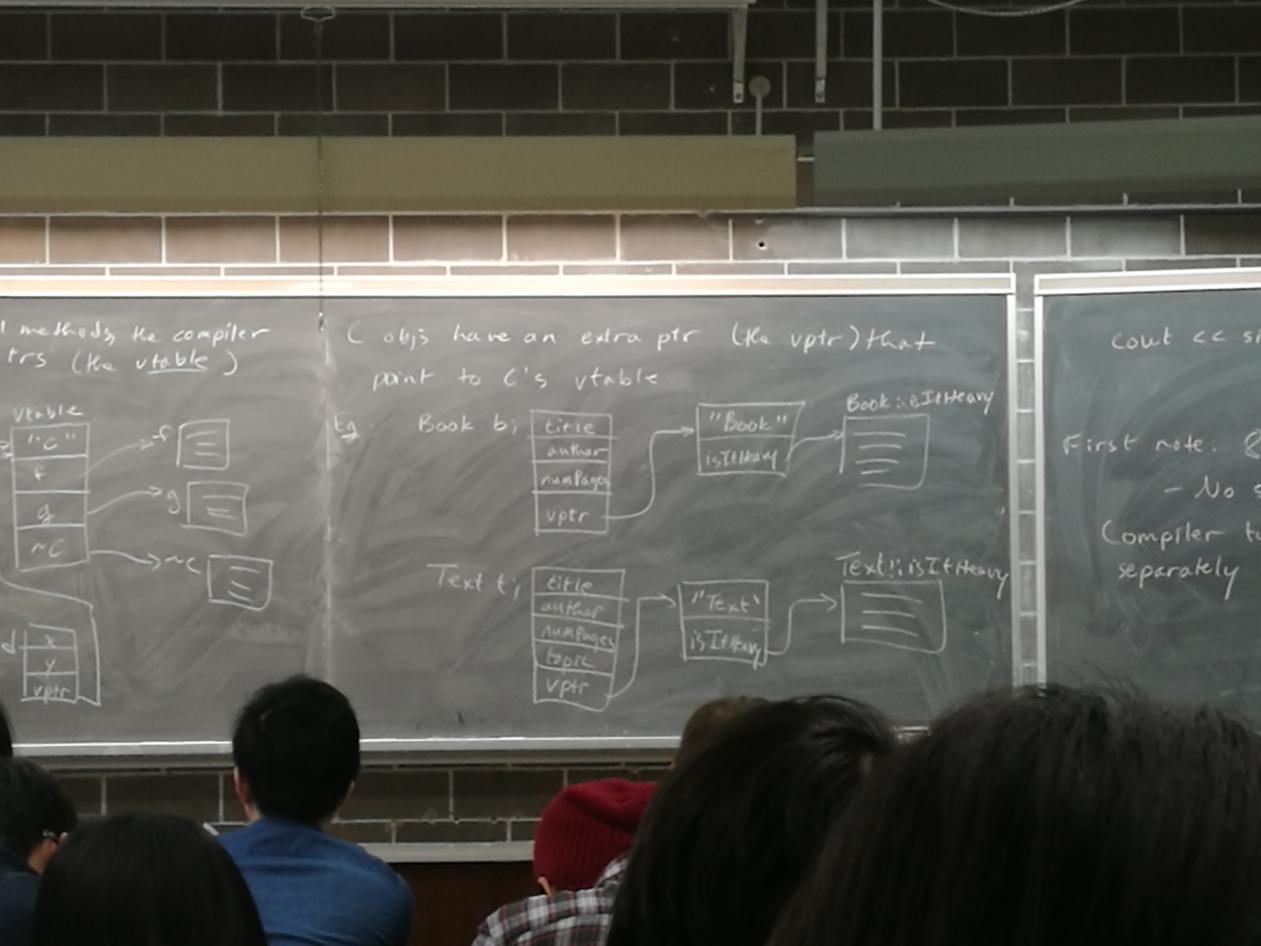
};

[]

C objes have an extra ptr (the vptr) that point to c’s vtable

Eg

Book b;

[]

Calling a virtual method:

- follow vptr to vtable

- fetch ptr to actual method from the table

- follow the f’n ptr & call the f’n // all, at run-time

so, Virtual method calls incur a small overhead cost

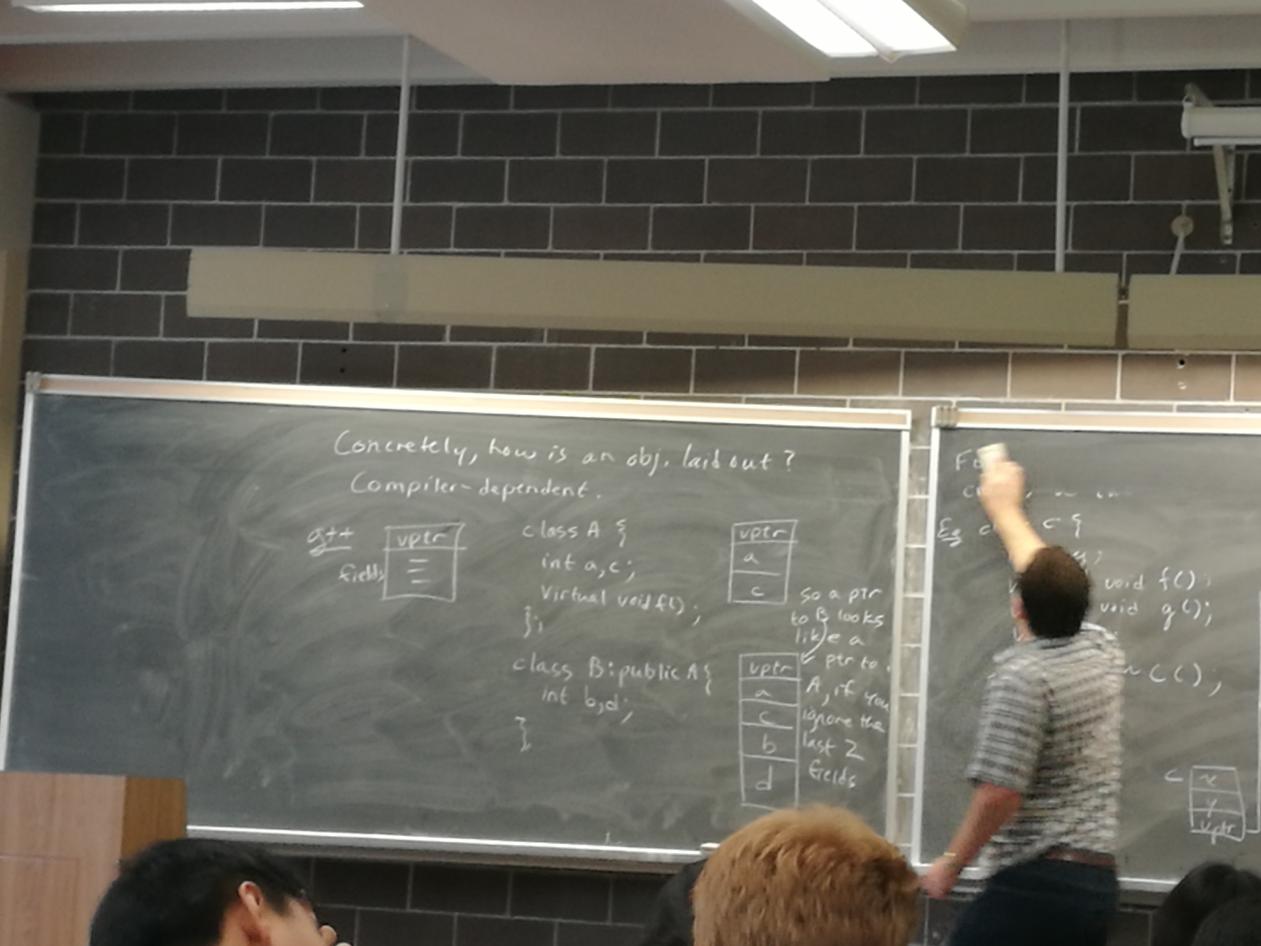
Also: declaring at least one virtual method adds a vptr to the object.

so, classes with no virtual methods produce smaller objects than if some methods were virtual

Concretely, how is an obj. laid out?

Compiler-dependent.

g++ fields

[]

Multiple Inheritance

A class can inherit from >1 parent.

class A {

public:

int n;

};

class B{

public:

int b;

};

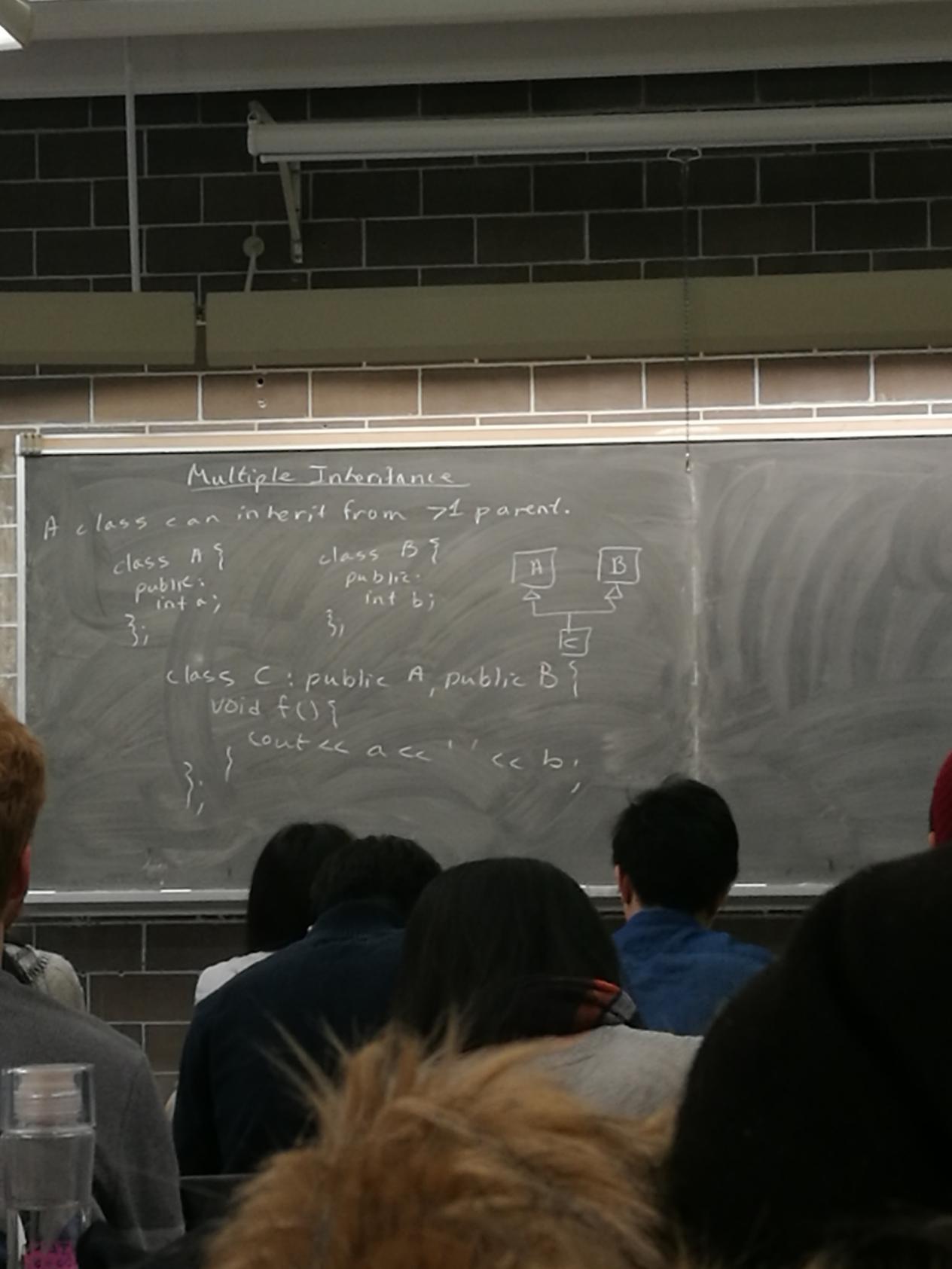
class C:public A, public B{

void f() {

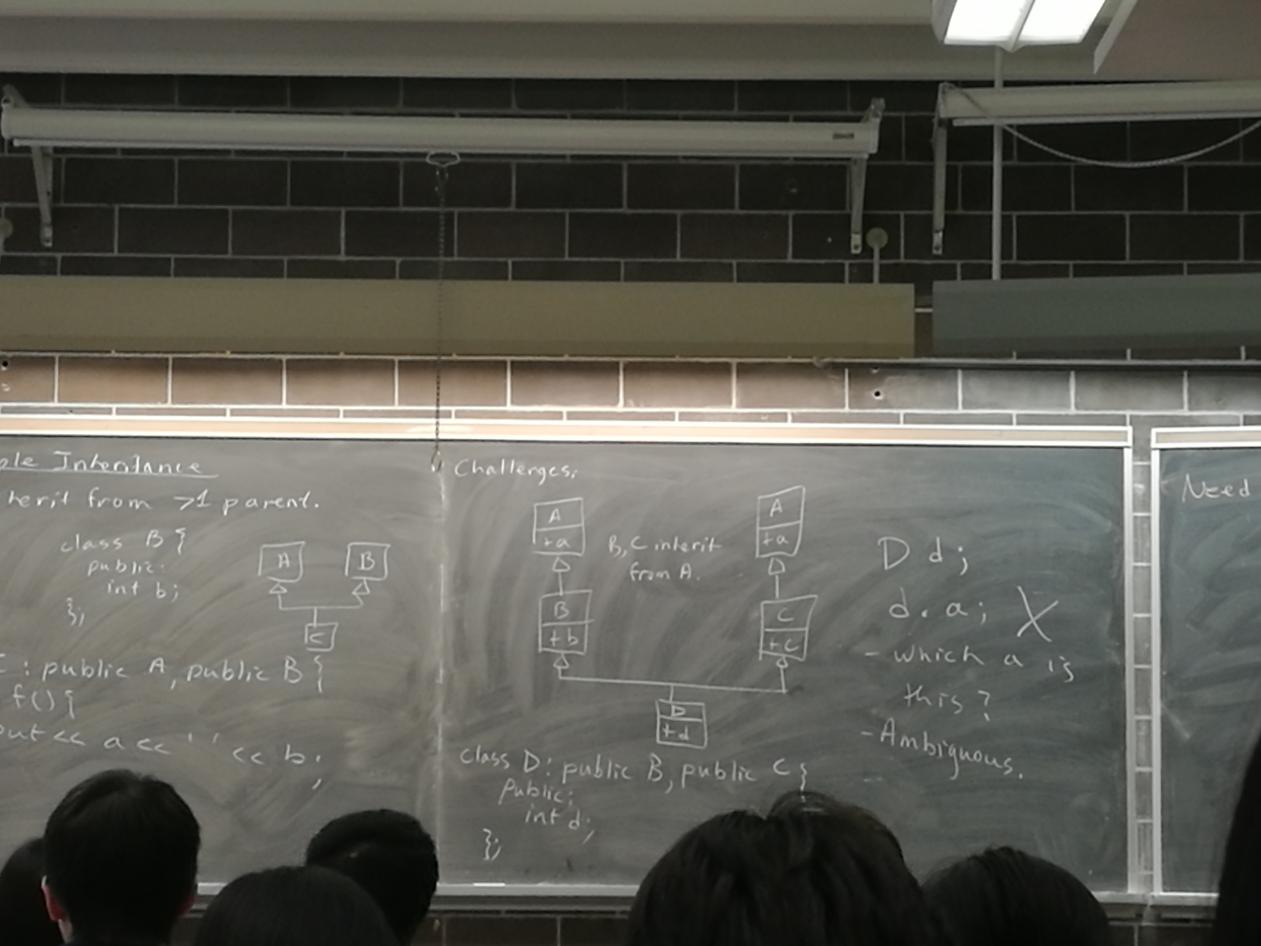
cout << a << ‘ ‘ << b;

}

};

[]

challenges:

[]

class D:public B, public C{

public:

int d;

};

D d;

d.a; X(false)

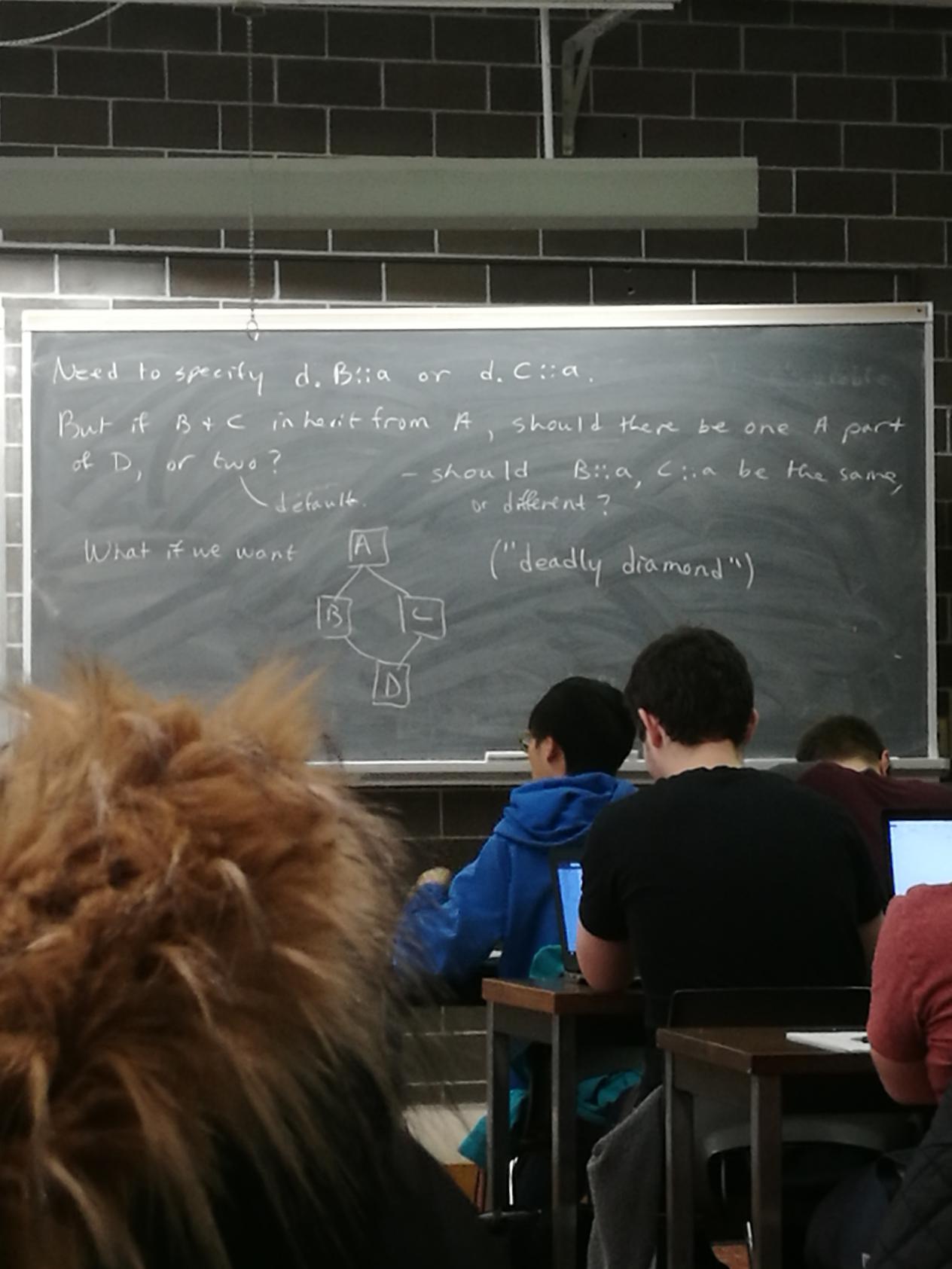
Which a is this?

-Ambiguous

Need to specify d.B::a or d.C::a

But if B + C inherit from A, should there be one A part of D, or two?

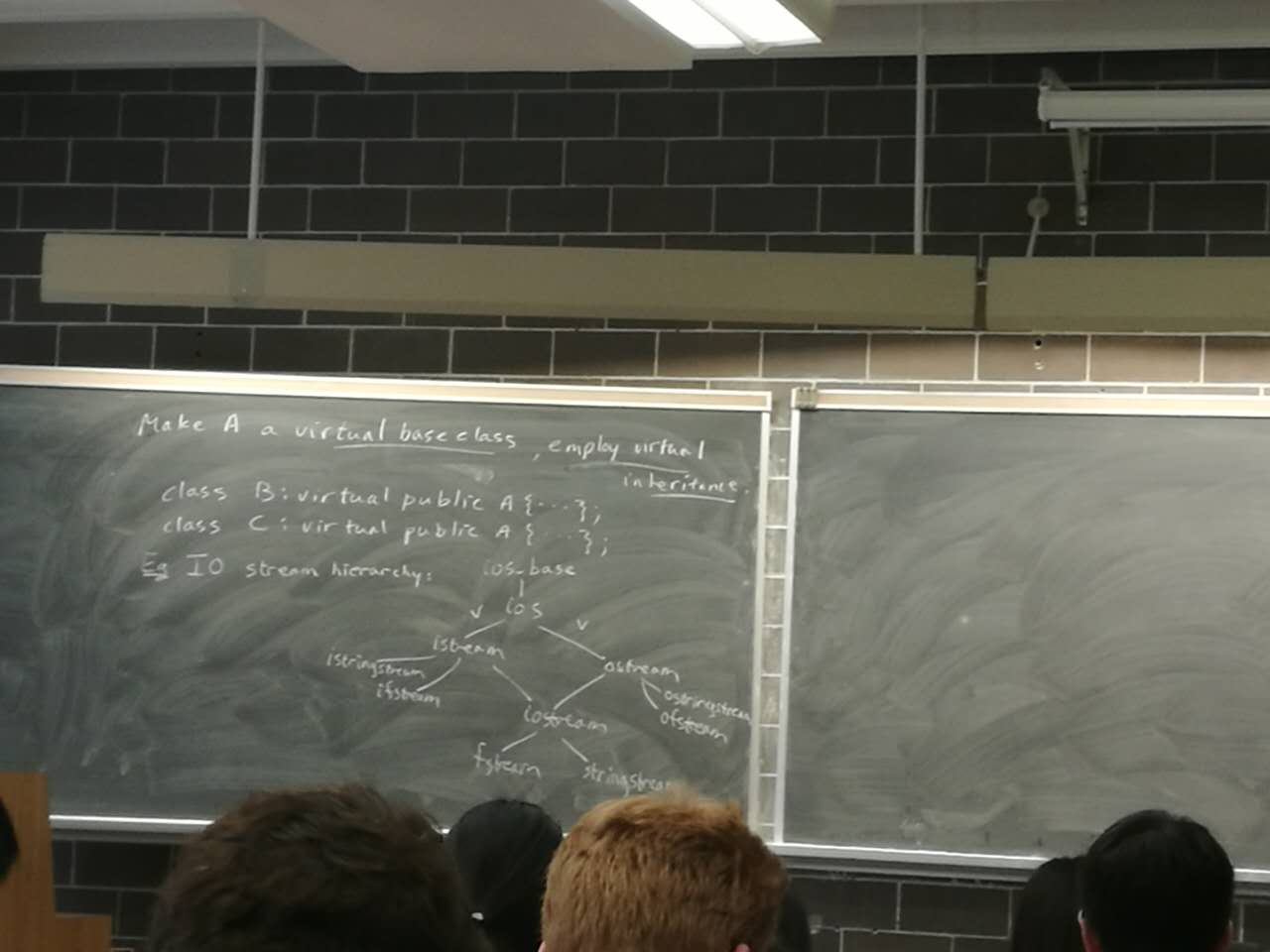
-default -should B::a,C::a be the same or different?

What if we want [] (“deadly diamond”)

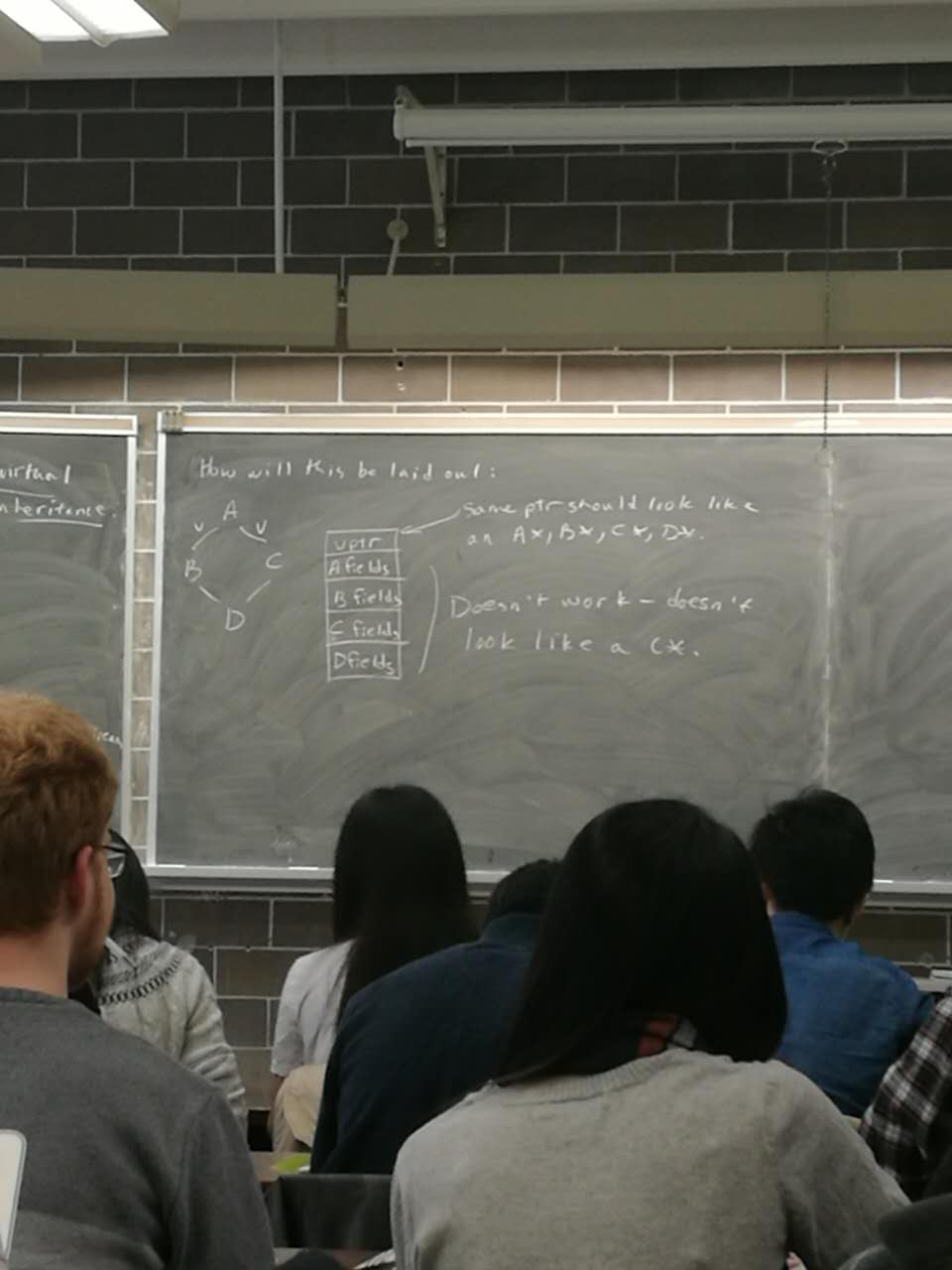
Make A a virtual base class, employ virtual inheritance

class B:virtual public A {...};

class C:virtual public A {...};

Eg IO stream hierarchy:[]

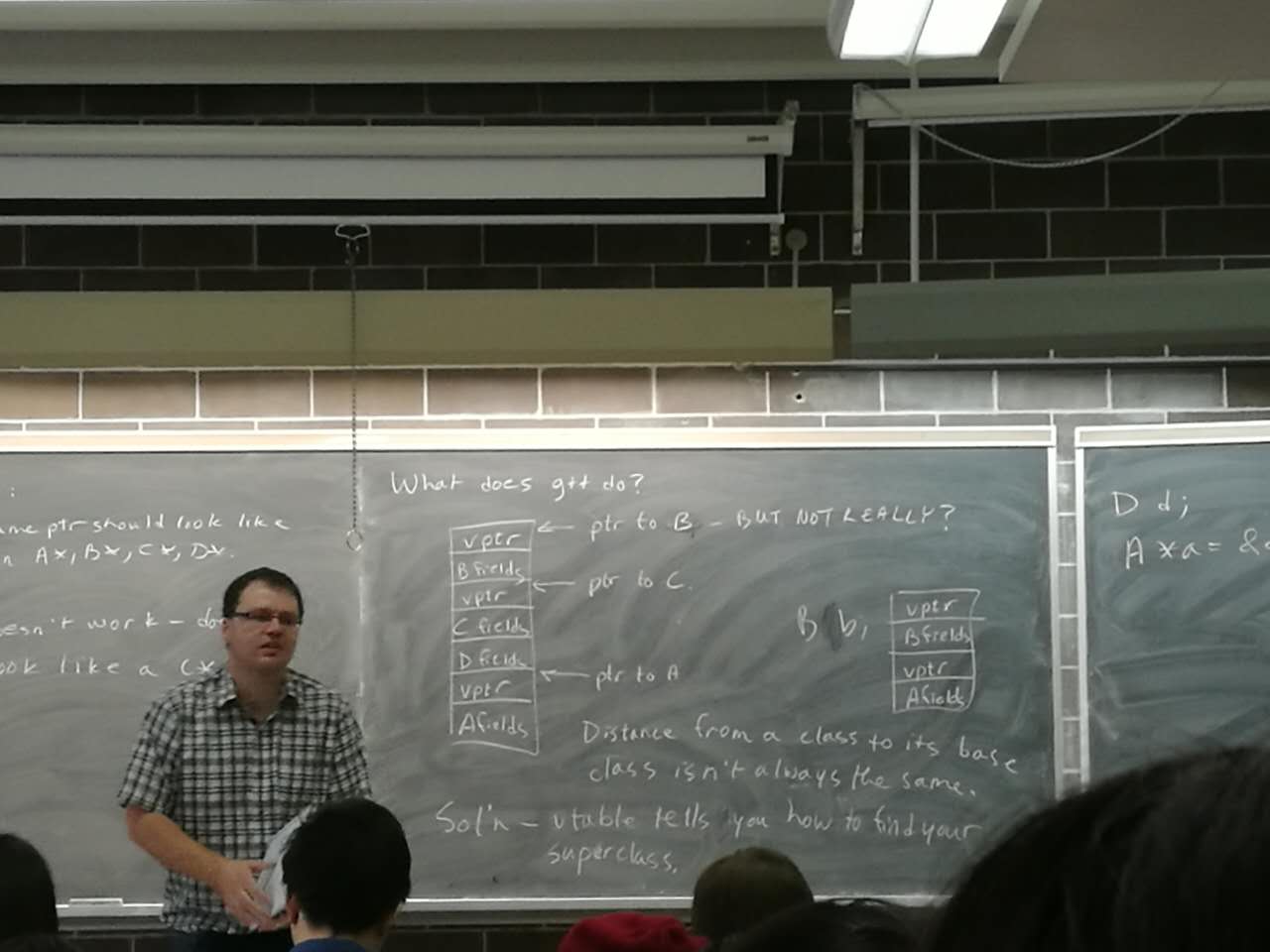
How will this be laid out:

[]

vptr:some ptr should look like an A\*,B\*,C\*,D\*

A B C D fields doesn’t work - doesn’t look like l C\*

What does g++ do?

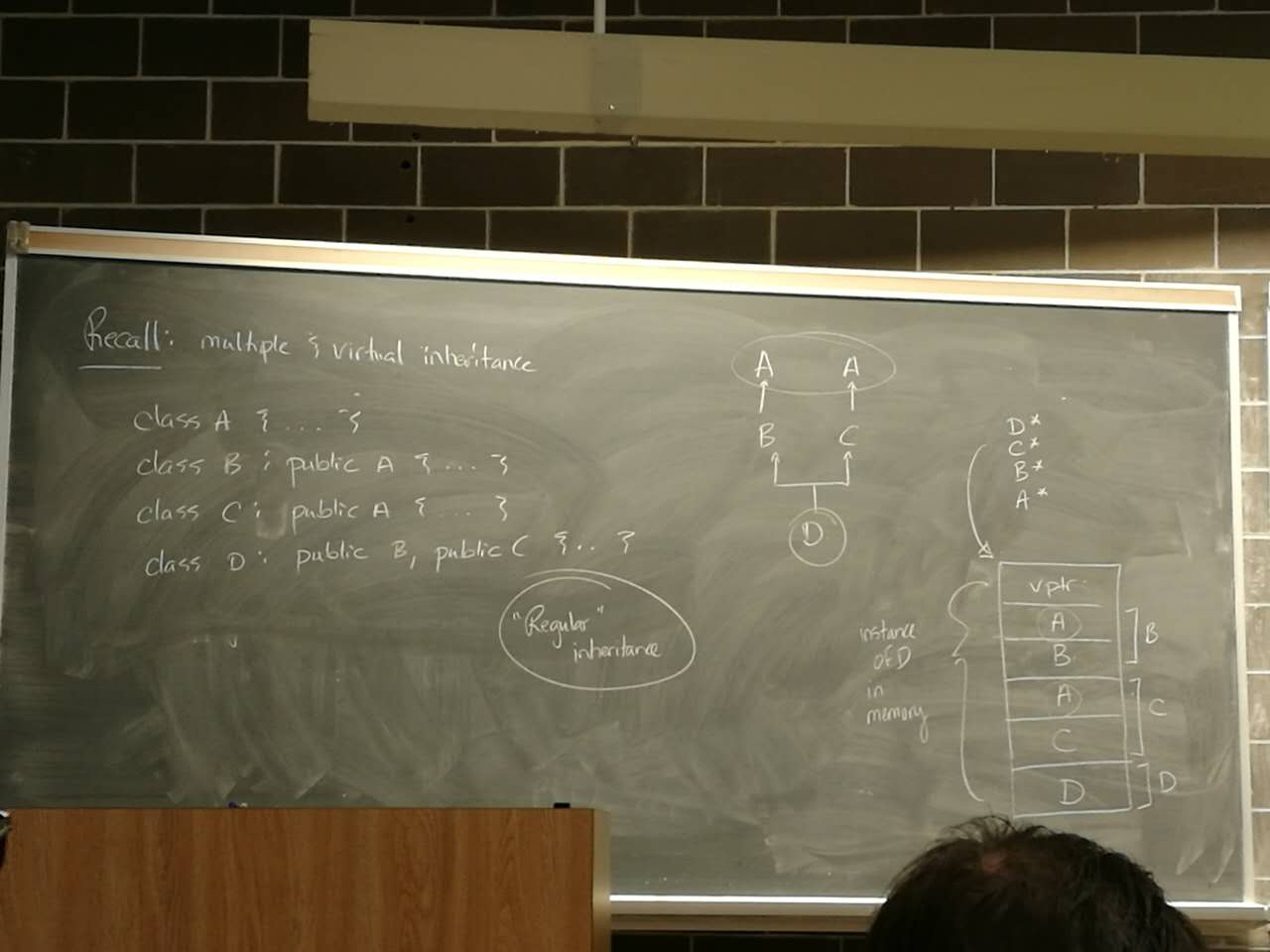
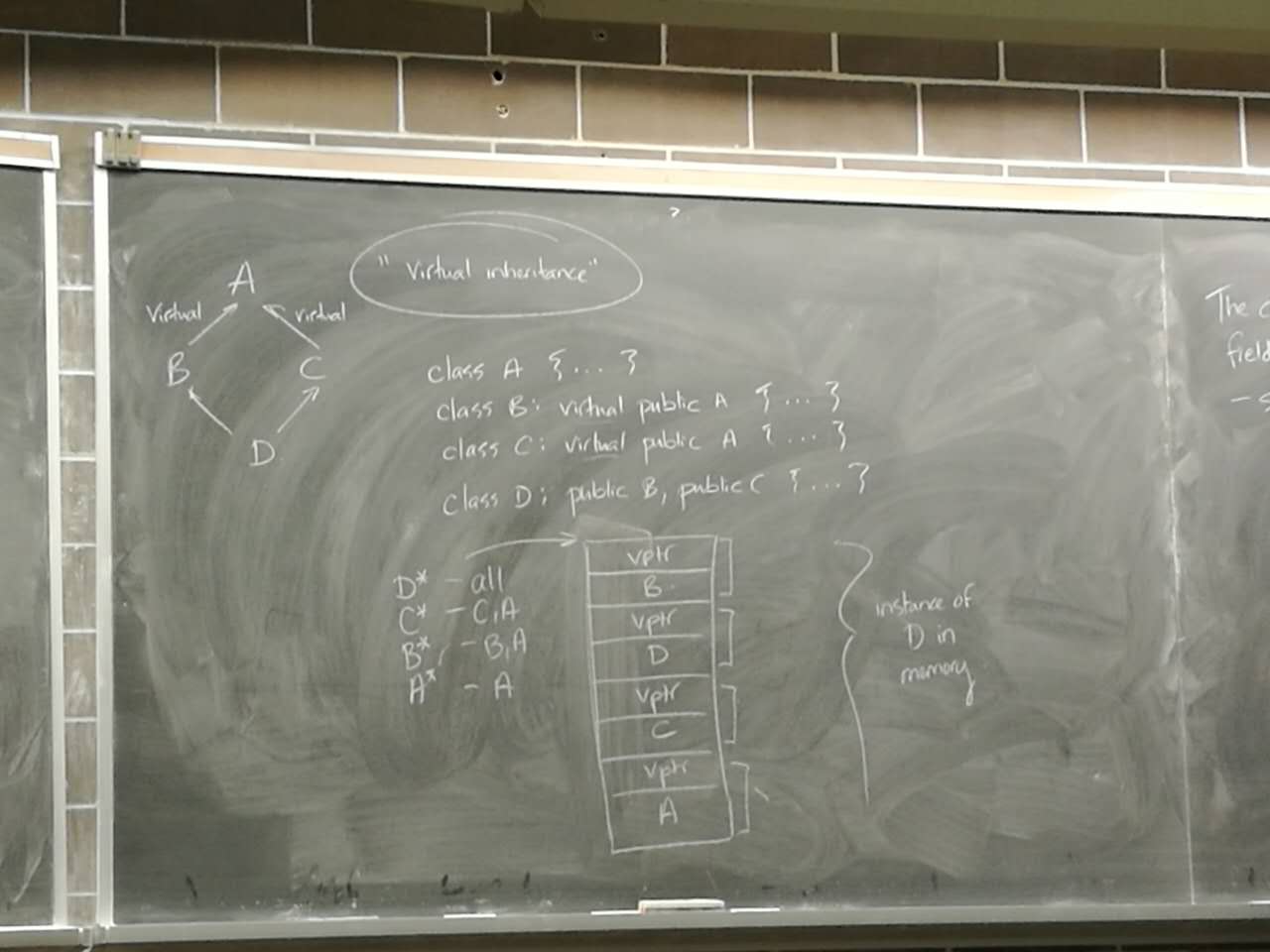
[]

Distance from a class to its base class isn’t always the same

Sol’n - vtable tells you how to find your superclass

D d;

A \*a = &d;

12.1--

The challenge is that a class only knows its own fields(and base class fields)

-so B should only be aware of B and A

-but “distance” between “B” and “A” fields depends on other classes

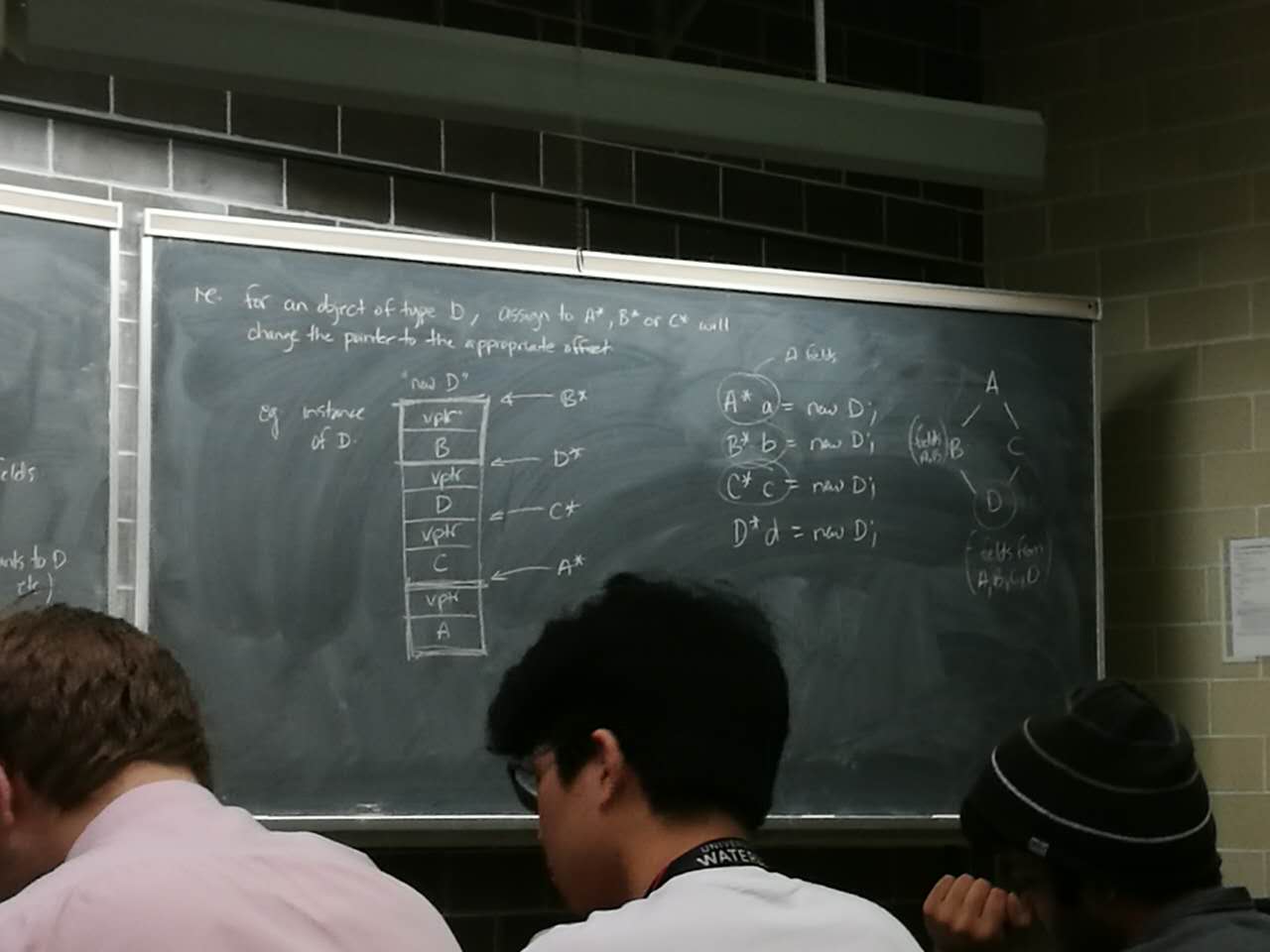
ie B\* can’t figure it out

Goal- A pointer of a particular type to find it’s fields and base class fields

Sol’n 1)derived class will have a pointer to it’s base class fields in it’s value(assume that a B\* points to B, D\* points to D etc.)

2) pointer assignment between A,B,C,D changes the address of the pointer

ie for and object of type D, assign to A\*,B\* or C\* will change the pointer to the appropriate offset

-

Template functions

- generating types for function parameters

template <typename T> T min(T x, T y) {

return x < y ? x:y;

}

eg. usage

int f() {

int x=1,y=2;

int z = min(x,y); //T = int

}

Note:usually no need to say min<int> - compiler can infer with template functions (will not infer w/ template classes)

if C++ is unable to determine the type(ie function has no arguments)

z = min<int>(x,y);

or char w = min(‘a’,’c’); // T = char based on types

auto f = min(1.0,3.0); //T = float

For what types can min be used? For what types will it compile?

- any type for which operator< is defined

Recall:

void for-each(AbstractIterator start, AbstractIterator finish, int (\*f) (int)) {

abstract base class

while(start != finish) {

f(\*start);

++start;

}

}

This works as long as:

-AbstractIterator supports !=, ++, \*

-f can be called as a function

So, we can generalize this, make types of parameters into template arguments:

template <typename Iter, typename Func>

void for-each(Iter start, Iter finish, Func f) {

while(start != finish) {

f(\*start); // implementation the same

++start;

}

}

Why?

Now it works with any type that supports !=,\*,++ -including raw pointers

eg

void f(int n) { cout << n <<endl;}

int a [] = {1,2,3,4,5};

for-each(a,a+5,f); // print the array

The algorithm library (STL)

A suite of template functions - many of which work w/ iterators . #include <algorithm>

“modern” C++ - built an iterators, template functions.

1. for-each - works as described
2. find - return an iterator to first match in a collection.

template < typename Iter,typename T>

Iter find(Iter first,Iter last, const T&val) {

// returns iterator to first match of val

// last if no match found

1. count - works like find, but returns number of matches
2. copy - copies elements between two containers

template < typename InIter, typename OutIter>

OutIter copy(InIter first, InIter last, OutIter result) {

//copies from range [first,last) to another, starting at result

//note - doesn’t allocate new memory, so output container

//must have space available

eg

vector<int> v {1,2,3,4,5,6,7};

vector<int> w (4); //allocate 4 ints;

copy(v.begin()+!,v.begin()+5,w.begin()); //w = {2,3,4,5};

5)

transform - execute a function over an iterator.

template<typename InIter, typename OutIter, typename Func>

OutIter transform (InIter first,InIter last,OutIter result, Func f) T

while(first != last) {

\*result = f(\*start);

++first;

++result;

}

return result;

}

eg

int add1(int n) {return n+1;)

...

vector<int> v {2,3,5,7,11};

vector<int> w (v.size());

transform(v.begin(),v.end(),w.begin(),add1); // w = {3,4,6,8,12}

---

How generalized is this?

1. what can we use for Func? - anything supports being called like a function
2. what can we use for InIter and OutIter? - !=,++,\*

So,how is f used?

- Just as f(\*start)

- we can use anything that can be used as a function

Function Object

ie. objects that behave like functions

- we can overload operator() for objects

eg

class Plus1 {

public:

int operator() (int n) {return n+1;}

};

eg

Plus1 p;

p(4); // produce 5;

ie works the same as our add1 function

eg

transform(v.begin(),v.end(),w.begin(),Plus1{}); // calls the default ctor

produce an object that supports obj()

Generalize:

class Plus {

int m;

public:

Plus(int m):m{m}{}

int operator() (int n) {return n+m;}

};

Now: transform (v.begin(),v.end(),w.begin(),Plus{2}); // increment by 2

Advantage of objects over functions?

-maintain state! // ie they have state

Instance of Plus, Plus1 are called function objects(also called functors(do not use on final))

eg

class IncreasingPlus {

int m = 0;

public:

int operator() (int n) {return n+(m++);}

void reset() {m=0;}

};

eg usage

vector<int> v (5,0); //five zeros

vector<int> w (v.size());

transform(v.begin(),v.end(),w.begin(),IncreasingPlus{}); // w = {0,1,2,3,4};

Lambdas

Q - how many ints in a vector are even?

Vector<int> v {...};

bool even(int n) { return n % 2 == 0;}

int x = count\_if(v.begin(),v.end(),even); //count # of the responses

Wow, self... that’s a lot of typing, for a function that I’ll never use again

Lambda:

int x count\_if(v.begin(),v.end(),[](int n) {return n % 2 == 0;});

lambda

Iterators

Anything that supports !=,++,\*

-we can apply the notion of an iterator to other things - like streams

eg #include <iterator>

vector <int> v {1,2,3,4,5};

ostream\_iterator <int> out {cout ,”,”};

copy(v.begin(),v.end(),out); // prints 1,2,3,4,5,

But consider:

vector<int> v {1,2,3,4,5};

vector<int> w;

copy(v.begin(),v.end(),w.begin()); // Wrong - no space allocate for w!

does not allocate memory for w

What if we have an iterator, whose assign operator did the insertion?

vector<int> v {1,2,3,4,5};

vector<int> w;

copy(v.begin(),v.end(),back\_inserter(w));

works!allocate row memory!