Lecture 11

Object Initialization, Copying, Destroying

Last time: initializing fields that are const/refs.

In-class initialization, not what we need:

```
struct Student {
    const int id = 20733115;
    int assns, mt, final;
};
```

Can't assign in constructor body.

Steps for Object Construction:

1. Space is allocated
2. Field initialization: call default constructors for fields that are objects
3. Constructor body runs

Let's "hijack" step 2 using Member Initialization List (MIL)

- syntax available only in constructors
- can be used to initialize all/some fields

```
Student::Student(int id, int assns, int mt, int final) :
    id{id}, assns{assns}, mt{mt}, final{final} {}
// field is first parameter, i.e. field = final, but not {final}
// second parameter points to declaration, i.e. {final} points to int final
```

- MIL always initializes fields in declaration order
- In some cases, using an MIL to initialize a field is more efficient than assigning it in the constructor body

```
struct Bla {
    Student s;
    Vec v;
};
```

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

```
Vec v{3,4};
```

MIL has priority over in-class initialization.

Copy Constructors

```
Student billy{80,50,70};
Student bobby{billy};
// Student bobby = billy;
```

Construct an object as a copy of an existing object

- We get one for free
- Copies fields of existing object into the new object

Every class comes with:

- default (0 param) constructor
- Big 5
  - copy constructor
  - destructor
  - copy assignment operator
  - move constructor
  - move assignment operator

A copy constructor takes a reference of the existing object.

```
Student::Student(const Student &other)
    : assns{other.assns}, mt{other.mt}, final{other.final} {}
```

The free copy constructor might not always do the "correct" thing.

```
struct Node {
    int data;
    Node *next;
    Node(int, Node *);
    Node(const Node &);
};
Node::Node(int data, Node *next) : data{data}, next{next} {}
Node::Node(const Node &other) : data{other.data}, next{other.next} {}
Node *np = new Node{1, new Node{2, new Node{3, nullptr}}};
// np is on the stack, {1, 2, 3} nodes are on the heap
Node n1{*np}; // Calls copy constructor
                          // Node n1 = *np;
Node *n2 = new Node{*np}; // copy constructor
struct llNode *np = malloc(sizeof(struct llNode));
```

Refer to https://github.com/RRethy/NOTES-CS246/blob/master/oct_11.md for diagrams.

Deep Copy Constructor

```
Node::Node(const Node &other) {
  data = other.data;
  // next = new Node{*other.next};
  if (other.next) next = new Node {*other.next};
  else next = nullptr; // Deep copy
}
```

```
Node::Node(const Node &other) :
    data{other.data},
    next{other.next ? new Node{*other.next}
                    : nullptr} {}
// A ? expr1 : (B ? expr2 : expr3)
```

A copy constructor is called

1. constructing object as a copy of an existing object
2. passing an object by value
3. returning an object by value

It is (2) which is the reason why a param to a copy constructor is a reference.

Constructors with one parameter

```
{explicit} Node::Node(int data) : data{data}, next{nullptr} {} // Explicit key word makes l
Node n{4};
void foo(Node);
foo(n);
Node = 4; // Valid
foo(10); // Valid
```

Single parameter constructors create implicit or automatic conversions. # Lecture 12

Big 5 (Continued)

Destructors

```
Node *np = new Node{1, new Node{2, new Node{3, nullptr}}};
delete np; // calls destructor on Node (1); Node (2) & (3) leak
```

- every class comes with a destructor
- destructor calls destructors on fields that are objects

```
Node::~Node() {
  delete next; // deleting a nullptr is safe
}
```

Copy Assignment Operator

```
Student billy{80,50,70};
Student bobby{billy}; // copy constructor
Student jane{...}
jane = billy; // copy assignment operator
jane.operator=(billy); // operator= is a method
```

Sometimes the one we get for free does not do the "correct" thing.

```
Node n1{...}, n2{...}, n3{...};
n2 = n1; // n2.operator=(n1);
n3 = n2 = n1; // n3.operator=(n2.operator=(n1))
```

```
Node &Node::operator=(const Node &other) {
  if (this == &other) return *this; // (2)
  data = other.data;
  delete next; // (1)
  next = other.next ? new Node{*other.next} : nullptr;
  return *this;
}
```

(1) `next` might already be pointing to heap memroy

```
n1 = n1; // n1.operator = (n1);
```

- `next` and `other.next` are the same
- accessing a dangling pointer

(2) Self-Assignment Check

- If `new` fails to allocate memory, `next` is not assigned.
- Makes `next` a dangling pointer
- Better to delay deleting next until we know that new will not fail

Perfect Implementation

```
Node &Node::operator=(const Node &other) {
  if (this == &other) return *this;
  Node *temp = next;

  // If this line fails, no lines after this will be run
  next = other.next ? new Node{*other.next} : nullptr;

  delete temp; // delaying the deleting to see if the above line fails
```

```
  data = other.data;
  return *this
}
```

Alternative Method - Copy & Swap Idiom

```
// node.h
struct Node {
  void swap(Node &other);
  Node &operator=(const Node &);
};
```

```
// node.cc
#include <utility>

void Node::swap(Node &other) {
  std::swap(data, other.data);
  std::swap(next, other.next);
}

Node &Node::operator=(const Node &other) {
  Node tmp{other}; // deep copy. Memory allocated on the stack
  swap(tmp);
  return *this;
}
```

Complicated Example

```
// pass-by-value
Node plusOne(Node n) {
  for (Node *p {&n}; p ; p = p->next) {
    ++p->data;
  }
  // return by value
  return n;
}

int main() {
  Node n {1, new Node{2, nullptr}};
    Node n2 {plusOne(n)}; // 6 calls to copy constructor
  cout << n << endl << n2 << endl;
}
```

- 2 calls to basic constructor
- 2 calls to copy constructor (to copy arguments)
- 2 calls to copy constructor (to return)

- Node n2{returned value}
- 2 calls to copy constructor (construct n2 as copy of returned value) # Lecture 13

Move Operations, Operator Overloading, Arrays of Objects

Move Operations

An rvalue reference is a reference to a temporary (an rvalue).

If a move constructor is available, it is called whenever we are constructing an object from an rvalue. If we implement any of the Big 5 (copy constructor, copy operators=, destructor, move operator=) you lose move constructor.

```
Node n{1, new Node{2, nullptr}};
Node n1{PlusOne(n)};
n = PlusOne(n1); // PlusOne(n1) is an rvalue

Node &Node::operator=(Node && other) {
  swap(other); // Node::swap(Node &)
  return *this;
}
```

Rule of 5

If you need to write a custom copy constructor or copy operator=, or destructor, or move constructor, or move operator= then you usually need to write all of them.

Copy/Move Elision

Under certain conditions, the compiler is allowed (not required) to avoid some copy/move constructor calls (even if these have side effects).

```
Vec makeVec() {return Vec{_,_};}
Vec v = makeVec(); // makeVec() is temp
```

- This would cause move/copy constructor to be called
- The compiler may directly construct the Vec in the space for var v.

```
Void foo(Vec v) {...}
// foo(5);
foo(makeVec());
```

- f no_elide_constructors

Operator Overloading: Methods or standalone functions

- operator= is/must be a method
- `n1 = n2; // n1.operator=(n2)`

```
Vec v1{1,2};
Vec v2{3,4};
Vec v = v1 + v2; // v1.operator+(v2)
v = v1 * 5; // v1.operator*(5)
v = 5 * v1; // 5 * operator*(v)

Vec Vec::operator+(const Vec &other) {
  return Vec{x + other.x}, y + other.y}; // implicitly, x == this->x
}

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

Vec operator*(const int, const Vec &);

ostream &Vec::operator<<(ostream &out) {
  out << x << "..." << y; // v.operator<<(cout);
  return out;
}

// Usage
Vec v...;
v << cout;
v2 << (v1 << cout)
v.operator<<(cout);
```

```
struct Vec{
    ...;
    Vec(int, int);
};

Vec v; // bad
Vec arr[3]; // bad
Vec *p = new Vec[3]; // bad
```

No default constructor. Allocating an array of objects requires default constructing the elements. # Lecture 14

Arrays/const methods/invariants/encapsulation

```
struct Vec {
    int x, y;
  Vec(int, int);
};
Vec arr[3];                          // Won't compile as default
Vec *arr = new Vec[3];  // constructors are gone
```

Options

1. Implement a default (0 parameter constructor)
2. Stack arrays: use C's array initializing syntax

   ```
   Vec arr[3] = {Vec{1,2}, Vec{3,4}, Vec{5,6}};
   ```

3. Instead of array of objects, create array of pointers to object

   ```
   Vec *arr[3];
   Vec **arr = new Vec*[3];

   arr[i] = new Vec{1,2};
   for (...) {
     delete arr[i];
   }
   delete []arr;
   ```

Const Methods

```
struct Student {
    int assns, mt, final;
    float grade() {
        return 0.4 * assns + 0.2 * mt + 0.4 * final;
    }
};
const Student billy{80,50,70};
cout << billy.grade(); // won't compile
```

Problem: grade does not promise to not change fields.

- We can promise this by declaring the method **const**
- Const objects can only call methods that are constant.
- Fix:
- ```
  struct Student {
    int assns, mt, final;
  ```
  ```

```
    float grade() const {
        return 0.4 * assns + 0.2 * mt + 0.4 * final;
    }
};
```

Invariants

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

Node n1{1, new Node{2, nullptr}};
Node n2{3, nullptr};
Node n3{4, &n1};
```

When n1, n2, n3 are out of scope, the destructor will run automatically. Next of n3 is a stack address; cannot call delete. `~Node` assumed next is either `nullptr` or points to heap

Invariant: some assumption/statement that must not be violated for program to function correctly

Stack class: Last In First Out

We need to hide information/implementation to protect invariants.

Encapsulation

- treat objects as capsules (black box)
- interact through provided method (interface)

Public/Private Labels

```
struct Vec {
    Vec(int, int); // default visibility is public
  private:
  int x,y; // not visible from outside class
  public:
  Vec operator+(const Vec &);
};

Vec Vec::operator+(const Vec &o) {
```

```
  return Vec{x + o.x, y + o.y};
}

int main(void) {
  Vec v{1,2}; // default visibility was public
  Vec v1 = (v + v); // rvalue // operator+ is public
  cout << v1.x v1.y; // Bad since x,y are private
}
```

Advice: at a minimum, keep all fields private. Make select methods public.

```
struct ... {}; // default visibility is public (inherent from C)
class ... {}; // default visibility is private
```

```
class Vec {
    int x,y; // private by default
    public:
    Vec(int, int);
    Vec operator+(const Vec &);
};
```

Node Invariant

We will wrap Nodes within a List class

```
// list.h
class List {
  struct Node; // declare Node as private/nested class
  Node *thelist = nullptr;
  public:
  void addToFront(int n);
  int ith(int i);
  ~List();
};
```

```
// list.cc
struct List::Node {
  int data;
  Node *next;
  Node(int, Node*);
  ~Node() {
    delete next;
  }
};

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

```
}

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

int List::ith(int i) { // assume i is in range
  Node *cur = thelist;
  for (int j = 0; j < i && cur; ++j, cur = cur->next);
  return cur->data;
} // Traversal is O(n^2)
```

Lecture 15

Iterator Design Pattern, Short Topics

Last Time

```
class List {
    struct Node;
  Node *theList = nullptr;
  public:
  void addToFront(int);
  int ith(int);
  ~List();
};
```

Problem: List traversal is $O(n^2)$.

Solution:

```
class List {
  Node * theList;
  Node * curr;
  public:
  void reset();
  int &getNextVal();
};

Node * curr = theList;
while (curr) {
    ...;
    curr = curr->next;
}
```

Iterator Design Pattern

- Good solutions to common design problems
- Keep track of how much of the list we have traversed

```
for (int *p = arr; p != arr+size; ++p) {
    ... *p ...;
}
```

We will create a new class (Iterator) that acts as a pointer into a List.

- Iterator will have access to Node

To Do List

- a way to create fresh Iterator to the start of List
- a way to represent an Iterator to the end of List
- Overload !=, ++ (unary), * (unary)

```
class List {
    struct Node {
    ...;
  };
  Node *thelist = nullptr;
  public:
  class Iterator {
    Node * curr;
    public:
    explicit Iterator(node *n) : Curr{n} {}
    // *p
        int &operator*() {
            return curr->data;
        }
        // ++p
        Iterator * operator++() {
          curr = curr->next;
          return *this; // (unary prefix)
        }
        // Bonus: p++
        Iterator * operator (int) {
          curr = curr->next;
          return *this;
        }
        // !=
        bool operator!=(const Iterator &other) {
          return curr != other.curr;
        }
```

```
  };

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

  Iterator end() {
    return Iterator{nullptr};
  }
  // Prev list methods
  void reset();
  int &getNextVal();
};
```

Client Code

```
int main(void) {
    List l;
  l.addToFront(3);
  // (2);
  // (1);
  for (List::Iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl; // prints 1,2,3
    *it += l; // changes list to 2,3,4
  } // Alt Below
}
```

```
auto x = y; // define x to be the same type as y
```

C++ has built-in support/syntax for the Iterator Design Pattern.

```
for (auto i : l) {
    cout << i < endl;
} // i is int (by value)

for (auto &i : l) {
  i += l;
} // i is int &
// ref to data fields
```

Range-Based For Loops

We can use a range-based for loop for a class, `MyClass`, if:

- `MyClass` has methods `begin`/`end`, which return objects of a class, say `Iter`.
- `Iter` must overload !=, * (unary), ++ (unary prefix)

To force clients to use begin/end, make `Iter` constructor private.

- If we make it private, `List::begin/List::end` will lost access (private really implies no access).
- The `Iter` class can declare `List` to be a `friend`.

```
class List {
  ...;
  public:
  class Iter {
    Node * curr;
    explicit Iter(Node * p) ...;
    friend class List;
    ...;
  };
};
```

Friendship breaks encapsulation.

Advice: have as few friends as possible. # Lecture 16

Accessors/Mutators, System Modelling + Relationships

Last time

Keep fields private!

Provide aggressors (getters) and/or mutators (setters) which are public.

```
class Vec {
  int x,y;
  public:
  int getX() const {return x;}
  int getY() const {return y;}
  void setX(int n) {x = n;}
  void setY(int n) {y = n;}
};
```

Suppose a class has private fields and no aggressors/mutators.

- Want to implement the I/O operators
  - as standalone functions

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

System Modelling

- Identify the abstractions/entities/classes
- Interaction/relationship between classes

UML - Unified Modelling Language

- Scratching the surface

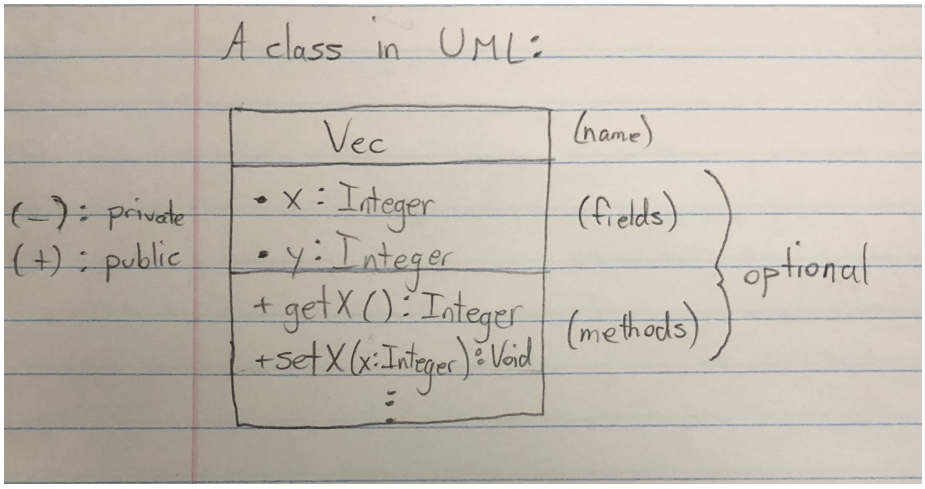


Figure 1: Class_UML

- Whenever possible use general types not specific to a language

Composition

```
class Vec {
    int x,y;
    public:
    Vec(int, int);
};
class Basis {
    Vec v1,v2;
};

Basis b; // won't compile as cannot default construct v1,v2
```

Solution

1. Provide default constructor for `Vec`.
2. Use the MIL to call on alternate constructor

```
Basis::Basis() : v1{0,1}, v2{1,0} {}
Basis b; // Valid; never default constructs v1,v2
```

- Embedding an object within another is composition.
- When an object takes ownership of another object.

Composition creates an OWNS-A relation

- Basis OWNS-A Vec

Typically, A OWNS-A B, if:

- When A is copied, B is copied
- When A is destroyed, B is destroyed

```
// List OWNS-A Node
class List {
    ...;
  Node * theList : ... {}
};
```

```
// Car OWNS-A Wheels
class Car {
    Wheel w1,w2,w3,w4;
};
```

```
// Is it still OWNS-A?
class Car {
  wheel *w1, *w2, *w3, *w4;
};
```

Aggregation

Parts in a Catalogue: Catalogue HAS-A Car-Part

Typically, A HAS-A B, if:

- When A is copied, B is not
- When A is destroyed, B is not

```
class Lake {
    Goose **geese;
};
```

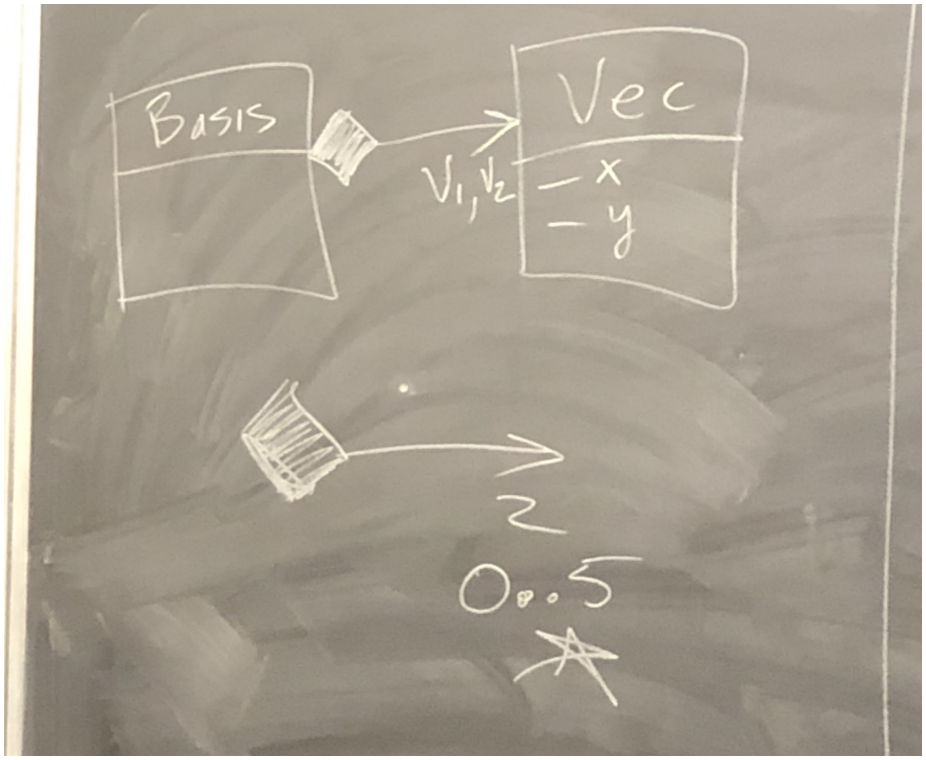


Figure 2: UML_2

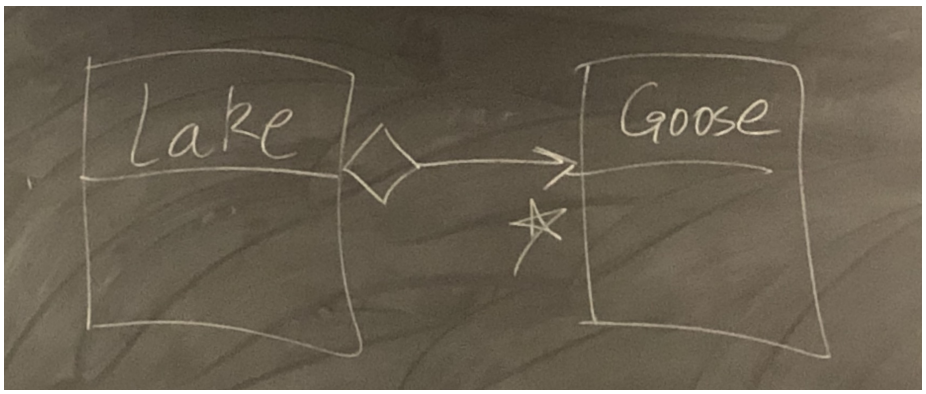
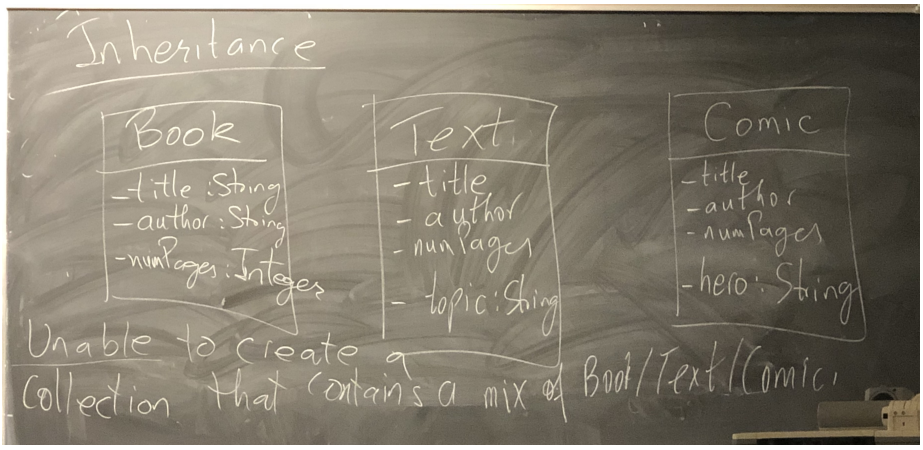


Figure 3: UML_3

Inheritance



Unable to create a collection that contains a mix of Book/Text/Comic.

Observation:

- Text IS_A Book (with an extra topic field)
- Comic IS_A Book (with an extra hero field)

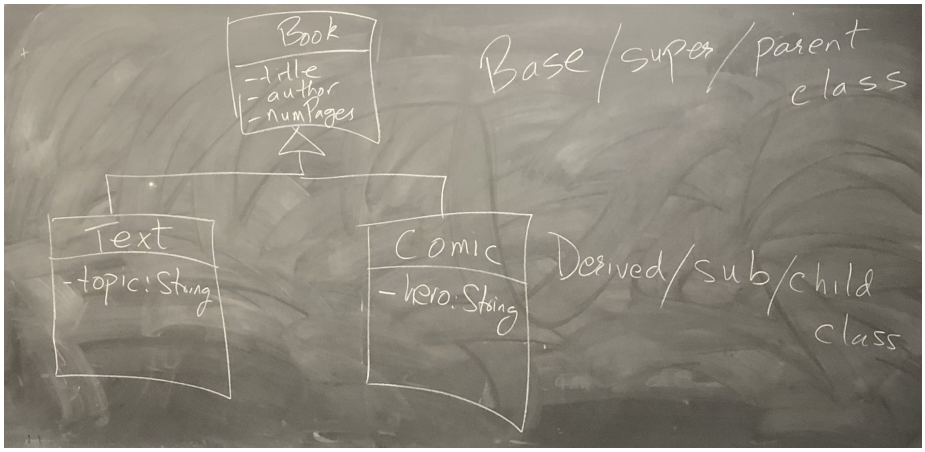


Figure 4: UML_5

Derived class inherits ALL (public & private) members (fields as methods) from the Base class.

Any method you can call on a Book object, you can call on a Text object.

- ifstream IS_A istream

Text inherits private fields: title, author, numPages

- Yet Text objects cannot access these fields.

```
Text::Text(string t, string a, int n, string topic) :
    title{t}, author{a}, numPages{n}, topic{topic} {}
```

Won't compile:

1. Inherited fields are also private.
2. MIL can only refer to fields declared in the class.
3. No default constructor for Book.

Lecture 17

Inheritance, Virtual Functions

Last Time

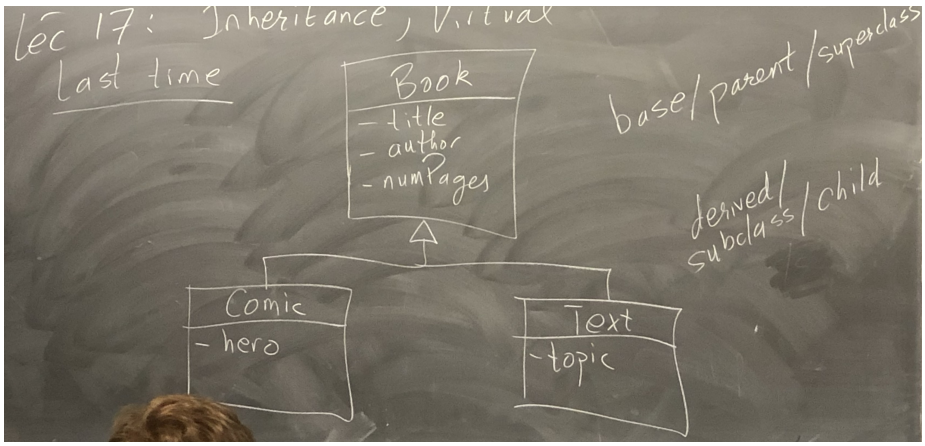


Figure 5: UML_6

- A derived class inherits all members from the base class
- A derived class cannot access private inherited members

```
class Book {
    string title, author;
    int numPages;
    public:
    Book(string, string, int);
};
```

```
class Comic : public Book {
  string hero;
  public:
  ...;
};

class Text : public Book {
  string topic;
  public:
  ...;
};

Text::Text(string t, string a, int n, string topic) :
    title{t}, author{a}, numPages{n}, topic{topic} {}
// Won't work because:
// (1) Inherited fields were private
// (2) MIL can only use fields declared by the class
// (3) No default constructor for Book *
```

4 Steps to Object Construction

1. Space is allocated
2. Superclass part is constructed
3. Subclass field constructed
4. Constructor body runs

```
Text::Text(string t, string a, int n, string topic) :
    Book{t, a, n}, topic{topic} {}
```

Protected

If a member is "protected" it is accessible by the class of its subclasses

```
class Book {
    protected:
  string title, author;
  int numPages;
  public:
  ...;
};

void Text::addAuthor(string a) {
  author += a; // Ok since author is protected
}

int main() {
```

```
  Text t = ...;
  t.author = ...; // Invalid
}
```

Protected breaks encapsulation

Advice: Keep fields private

- provide protected accessors/mutators
- UML: #protected

Method Overwriting

`isHeavy()?`

- Book isHeavy numPages > 200
- Text isHeavy numPages > 500
- Comic isHeavy numPages > 30

```
class Book {
    int numPages;
  public:
  int getNumPages() const { return numPages; }
  bool isHeavy() const { return numPages > 200; }
};

bool Text::isHeavy() const { return getNumPages() > 500; }
bool Comic::isHeavy() const { return getNumPages() > 30; }

Book b{..., ...,  50};
b.isHeavy(); // Book.isHeavy, false

Comic c{..., ..., 40, "batman"};
c.isHeavy(); // Comic::isHeavy, true

Book b1 = Comic{..., ..., 40, "batman"}; // Calls Book's copy constructor
b1.isHeavy(); // Book::isHeavy, false => Good
// Comic::isHeavy, true => Bad
```

Using superclass pointers, to point to subclass objects will not cause slicing.

```
Comic c{..., ..., 40, "Batman"};
Book *bp{&c};
bp->isHeavy();
// (1) Book::isHeavy()
// (2) Comic::isHeavy()
```

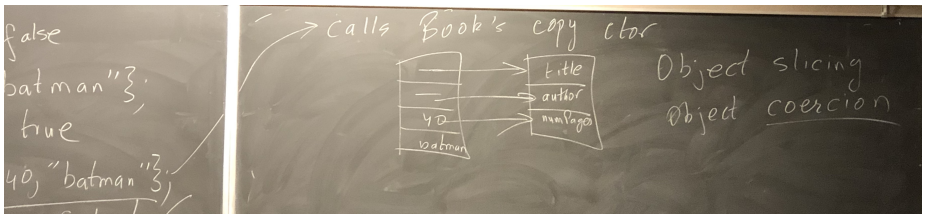


Figure 6: UML_7

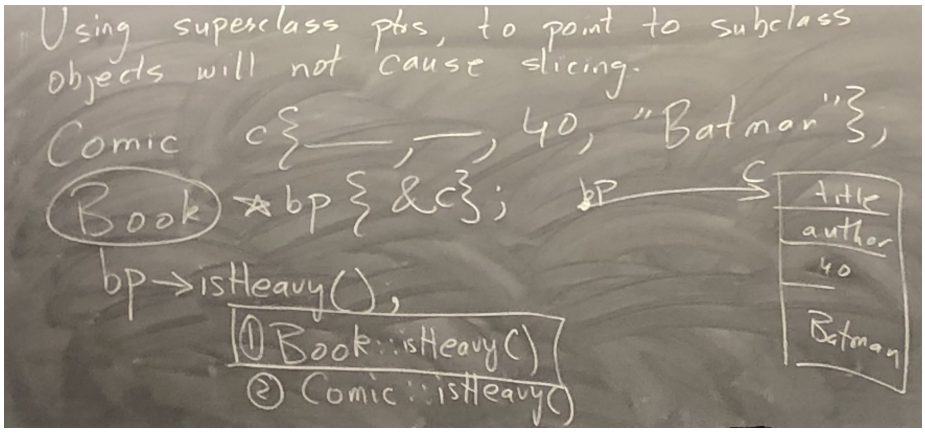


Figure 7: UML_8

By default, the compiler looks at the declared type of the pointer to choose which method to run (static dispatch). We typically want the method to be chosen based on the type of object.

Use "virtual" keyword

```
class Book {
    ...;
    public:
  virtual bool isHeavy() const { ...; }
};

bool Text::isHeavy() const override { ...; }
bool Comic::isHeavy() const override { ...; }

Comic c{..., ..., 40, ...};
Comic *cp{&c};
Book *bp{&c};
Book &br{c};

// Comic::isHeavy runs for all three
cp->isHeavy();
bp->isHeavy();
br.isHeavy();
```

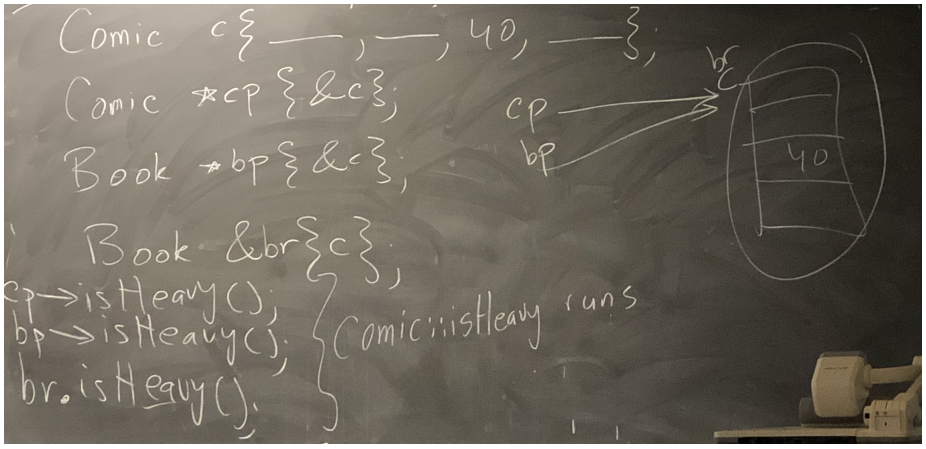


Figure 8: UML_9

Virtual Method

The choice of method is based on the runtime type of the object.

DYNAMIC DISPATCH

- More costly than static dispatch

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

Polymorphism: the ability to accommodate multiple types within one abstraction

```
istream & operator>>(istream &, ...);
// cin >> x;
// ifstream f{...};
// f >> x;
// ifstream IS-A istream
```

Destructors (4 steps to destroying objects)

1. Subclass destructor runs
2. Destructor for subclass fields that are objects
3. Superclass destructor runs
4. Space is reclaimed

Examples

```
class x {
    int *x;
    x(int m) : x{ new int[m] } {}
  ~x() { delete[] x; }
};

class y : public X {
  int *y;
  y(int m, int n) : x{m}, y{ new int[n] } {}
  ~y() { delete[] y; }
};

x *myEx = new y{10,20};
```

```
delete myEx; // Leaks memory
// ~y is never called:
//  Advice: All classes that can be subclassed should have a virtual destructor
```

Lecture 18

Abstract Classes, Templates, Exceptions

Last time: Destructors

```
class x {
    ...;
  virtual ~x() {...;}
};

class y : public x {
  ...;
  ~y(){...;}
};

x * myX = new y(...);
delete myX;
```

If a class will never have a subclass, declare it final.

```
class y final : public x {
    ...;
};
```

```
class Student {
    ...;
    public:
    virtual int fees() = 0; // Pure virtual method
};

class regular : public Student {
  ...;
  public:
  int fees() override {...;}
};

class coop : public Student {
  ...;
  public:
```

```
  int fees() override {...;}
};
```

`Student::fees()` has no implementation.

- We don't want to implement it
- Make it Pure Virtual (P.V.)

Virtual: subclasses ***may*** override behaviour

PV: subclasses ***must*** implement method to be ***CONCRETE***

Student has a PV method:

- It is incomplete
- It is an ABSTRACT class

A class is abstract if:

- It declares a PV method
- It inherits PV method(s) that it does not implement

A class is concrete if it is not abstract. Cannot create objects of an abstract class. `Student s; // bad`

Abstract Classes

- organize subclasses
- members (fields/methods) that are common across all subclasses can be placed in the base class
- act as an interface
- Polymorphism

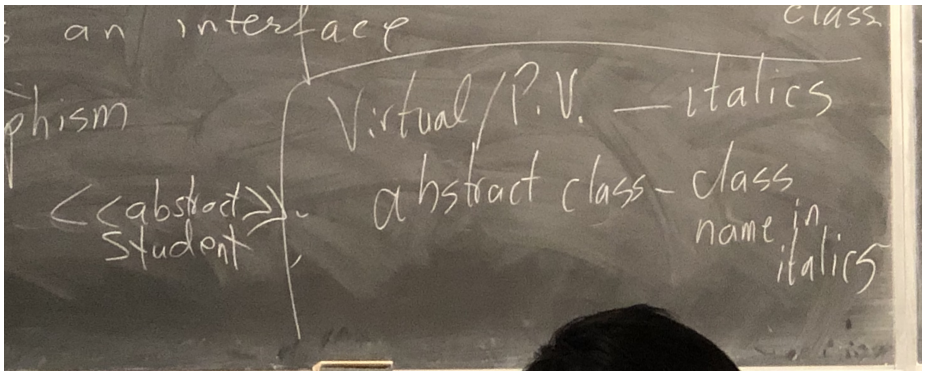


Figure 9: CODE_1

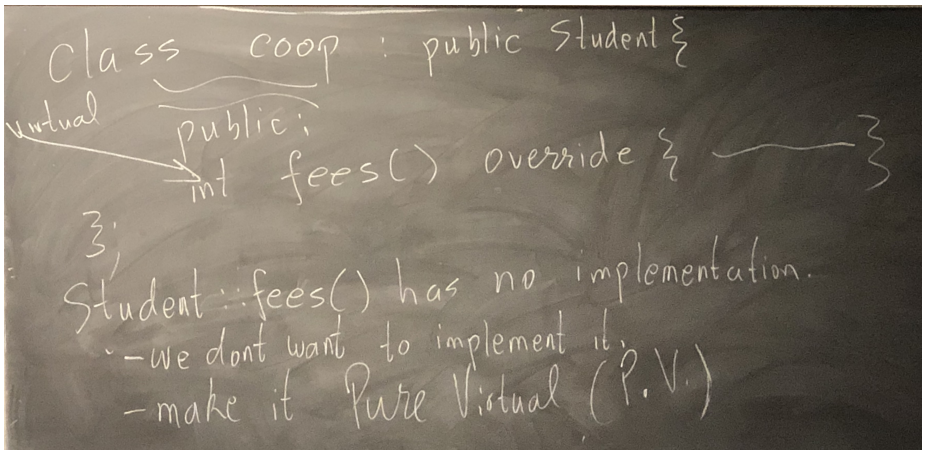


Figure 10: CODE_2

C++ Templates

```
template <typename T>

class Stack {
    int len;
  int cap;
  T * contents;
  public:
  void push T;
  void pop(void);
  T top();
  ~Stack();
};
```

C++ template class is a class parameterized on a type.

```
Stack<int> s1;
s1.push(1);

Stack<string> s2;
s2.push("hello");
```

Template List Class

```
template <typename T>

class List {
    struct Node {
```

```
        T data;
        Node *next;
    };
  Node * theList = nullptr;
  public:
  class Iterator {
    Node * cur;
    explicit Iterator(...;) : ...;
    public:
    T &operator*() {...}
    Iterator &operator++() {...}
    bool operator!=(...) {...}
    friend class List<T>;
  };
  T &ith(int i);
  void addToFront(T &t);
  ~List();
};
```

```
List<int> l1;
l1.addToFront(1);
l1.addToFront(2);

List<List<int>> l2;
l2.addToFront(l1);
```

Standard Template Library (STL)

`std::vector` `#include <vector>`

- template class
- dynamic length arrays
  - heap allocated
  - automatically resize as needed

```
vector<int> v{3,4}; // [3,4]
v.emplace_back(5); // [3,4,5]
v.pop_back(); // [3,4]
for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;
}
for (auto n : v) {
  cout << n << endl;
```

```
}
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
  cout << *it << endl;
}

v.erase(v.begin());
v.erase(v.endl() - 1);

v[i] // unchecked access
v.at(i) // checked access
```

If `i` is not in range, `out_of_range` exception is thrown

`lectures/c++/exceptions`

- `rangeError.cc`
- `rangeErrorCaught.cc`
  - After an exception is caught, program continues after the catch block.
- `callChain.cc`

Stack unwinding: When an exception occurs, the call stack is repeatedly popped until an appropriate catch block is found.

Lecture 19

Exceptions (cont'd), Design Patterns

Last Time: `callchain.cc`

Exception recovery can be done in stages

```
try {...}
catch (someExn e) { // catches someExn or a subclass
    // do part of recovery
  throw otherExn{...};
}
```

1. Throw some other exception
2. `throw e; // throw the caught exception`
3. `throw; // throws original exception`

All C++ library exceptions inherit from `std::exception`. C++ does not require exceptions to inherit from `std::exception`. In C++ you can throw anything.

Good Practice: Throw objects of existing exceptions or create your own exception classes

```
class BadInput {};
```

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

try {
    foo();
} catch(BadInput &) {
    n = 0;
}
```

Advice: catch by reference:

- out_of_range
- bad_alloc
- length_error

Observer Pattern

Publish/subscribe model

- Publish: Subject aggregates data
- Subscribe: Observer wants to be notified of new data

Decorator Pattern

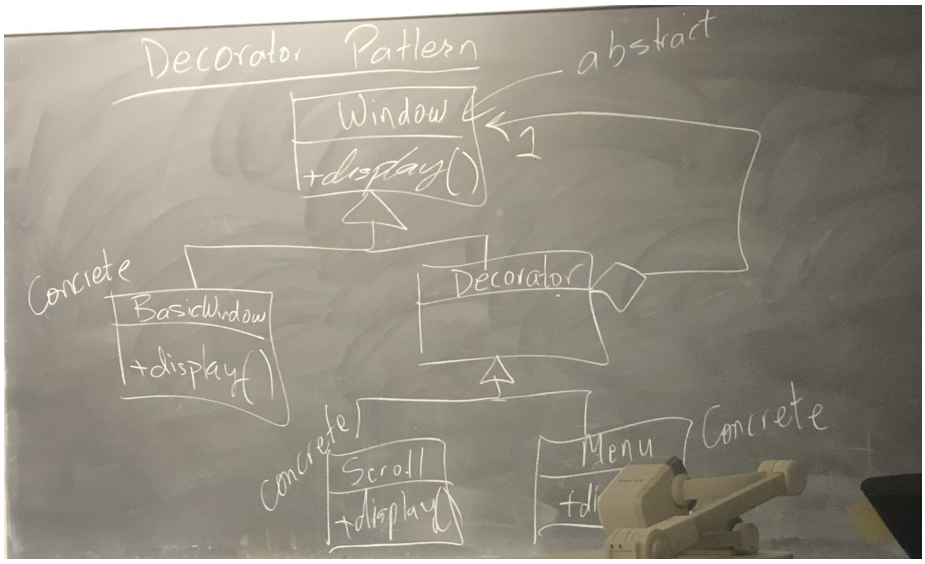


Figure 11: DECOR_1

- Scroll IS-A Window

- Scroll HAS-A Window

```
Window *w = new BasicWindow();
w = new Scroll{w};
w = new Menu{Scroll};
w->display();
```

Lecture 20

Object Oriented Design Philosophy

- Use abstract base classes to provide an interface
- Use pointers to base class that call those interface methods
- Behaviour changes based on the runtime of objects

Iterator Pattern

- `operator *`, `operator++`, `operator!=`

```
class AbsIter{
  public:
  virtual int &operator*() const = 0;
  virtual AbsIter &operator++() = 0;
  virtual bool operator!=(const AbsIter &) const = 0;
  virtual ~AbsIter() {}
};

class List {
  ...
  public:
  class Iterator : public AbsIter {
    ...
  };
  ... // List::Iterator IS-A AbsIter
};

class Set {
  ...
  public:
  class Iterator : public AbsIter {
    ...
  };
  ... // Set::Iterator IS-A AbsIter
};
```

We can now implement code that operates using `AbsIter` and not being tied to a specific data structure.

```
template <typename Fn>
void foreach(AbsIter & start, const AbsIter & end, Fn f) {
  while(start != end) {
      f(*start);
      ++start;
  }
}

// Example
void addTen(int &n) {
  n = n + 10;
}

List * l: ...
...
List::Iterator i = l.begin();
foreach(i, l.end(), addTen);
```

Factory Method Pattern

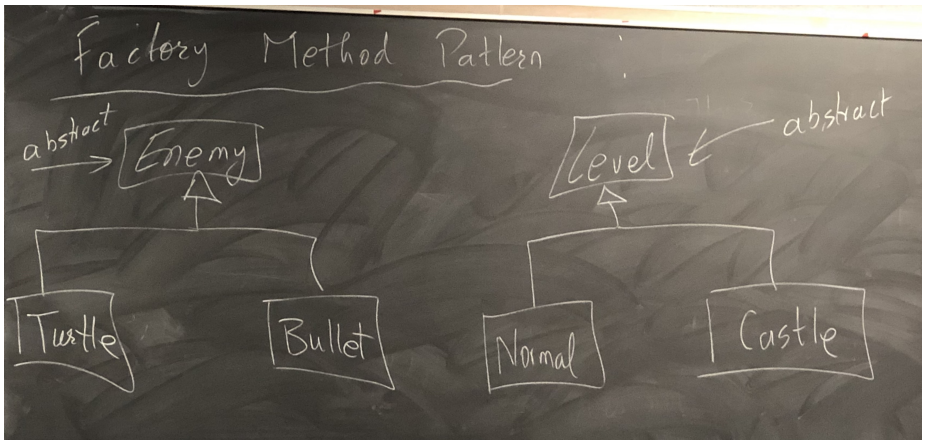


Figure 12: Factory_Pattern

```
Player * p = ...;
Level * l = ...;
Enemy * e = ...;

while (p->notDead()) {
```

```
    // generate enemy should depend on Level
  e = l->createEnemy();
    // attack player
}

class Level {
  public:
  virtual Enemy * createEnemy() = 0;
};

class Normal : public Level {
  public:
  Enemy * createEnemy() override { /* more turtles */}
};

class castle : public Level {
  public:
  Enemy * createEnemy() override { /* more bullets */ }
};
```

The factory method pattern is also called the "virtual constructor" pattern.

- `List::addToFront`
- `List::begin` / `List::end`

Template Method Pattern

- Base class implements the template/skeleton and the subclass fills the blanks.
- Base class allows overriding of some virtual methods, but other methods must remain unchanged (non-virtual).

```
class Turtle {
  ...
  void drawHead() {...}
  void drawFeet() {...}
  virtual void drawShell() = 0;

    public:
    void draw() {
        drawHead();
    drawShell();
    drawFeet();
    }
};
```

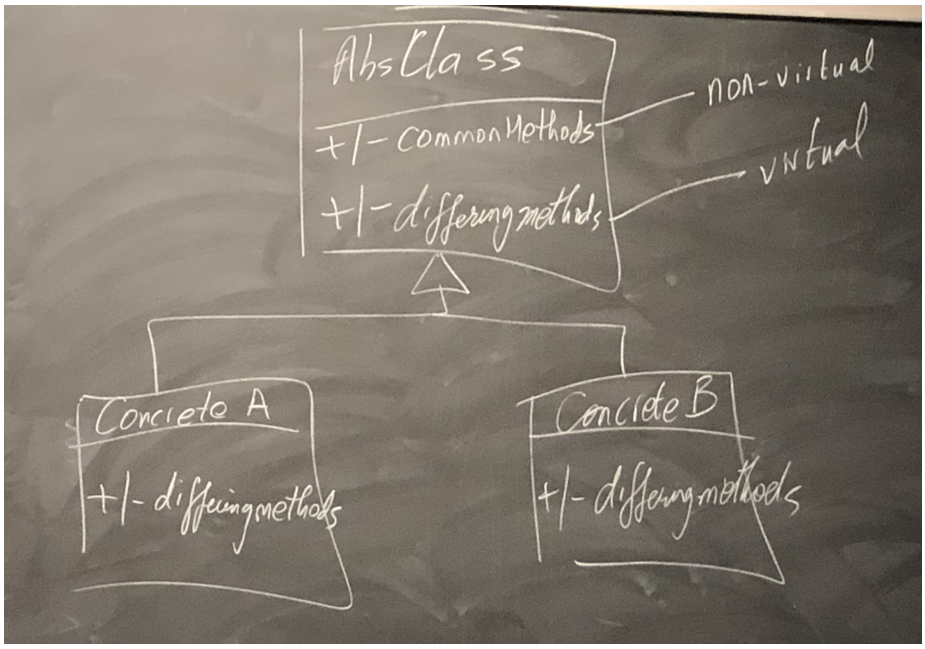


Figure 13: Template_Pattern

```
class RedTurtle : public Turtle {
  void drawShell() override {...}
};

class GreenTurtle : public Turtle {
  void drawShell() override {...}
};

Turtle *t = ...;
t->draw();
```

Non-Virtual Interface (NVI) Idiom

Public Virtual Method

- Public(interface) - provide a contract
- Virtual - invitation to subclasses to change behaviour
- ==Contradictory!==

In a NVI

- all public methods are non-virtual (except destructor)

- all virtual methods are private/protected

```
// Not using NVI
class Media {
    public:
    virtual void play() = 0;
};

// With NVI
class Media {
  virtual void doPlay() = 0;

  public:
  void play() { // CopyrightCheck();
    doPlay();
  }
};
```

Visitor Design Pattern

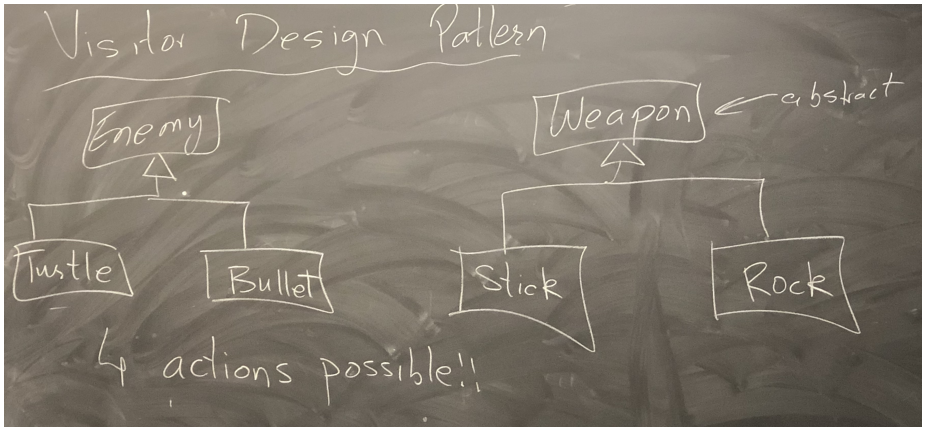


Figure 14: Visitor_Pattern

```
class Enemy {
    ...
    virtual void strike(Stick &) = 0;
    virtual void strike(Rock &) = 0;
};

Enemy *e = l->createEnemy();
Weapon *w = p->chooseWeapon();
e->strike(*w); // Won't compile
```

In C++, dynamic dispatch does not account for runtime type of parameter.

```
class Enemy {
    public:
    virtual void strike(Weapon &) = 0;
};

class Turtle : public Enemy {
    public:
    void strike(weapon & w) override { w.useOn(*this); } // Turtle
};

class Bullet : public Enemy {
    public:
    void strike(weapon & w) override { w.useOn(*this); } // Bullet
};

class Weapon {
  virtual void useOn(Turtle &) = 0;
  virtual void useOn(Weapon &) = 0;
};
```

Double Dispatch

- using a combination of overriding and overloading # Lecture 21

Compilation Dependencies

Last Time: Visitor Design Pattern (VDP)

- In C++, dynamic dispatch does not consider the runtime type of paramters
- **Double Dispatch:** combination of overriding and overloading

Book Hierarchy

- Want to add functionality that depends on runtime type of objects
- What if we don't have the source code?
- What if the behaviour does not really belong to the classes?
- We can do this if the Book hierarchy will "accept" BookVisitors

```
class Book { // Enemy
  public:
  virtual void accept(BookVisitor &v) { // strike(weapon)
    v.visit(*this); // useOn
```

```
  }
};

Class Text : public Book {
  public:
  void accept(BookVisitor &v) override { v.visit(*this); }
};

Class Comic : public Book {
  public:
  void accept(BookVisitor &v) override { v.visit(*this); } // this: Comic
};

class BookVisitor {
  public:
  virtual void visit(Book &) = 0;
  virtual void visit(Comic &) = 0;
  virtual void visit(Text &) = 0;
  ~BookVisitor();
};
```

Cataloging Books

- author -> # of Books
- topic -> # of Texts
- hero -> # of Comics
- string -> int

STL -> `std::map`

Template class, generalizes arrays

- Parameterized on 2 types -> Key/Value
- Key Type must support `operator<`
- ```
  std::map<string, int> m;
  m["abc"] = 5;
  m["def"] = 2;
  m.erase("abc");
  m.count("abc");

  for (auto p : m) { // std::pair<string, int>
    cout << p.first << p.second;
  } // Iteration is in sorted key order
  ```
  ```

```
Class Catalog : public BookVisitor {
    map<string, int> cat;
  public:
  void visit(Book &b) override { ++cat[bgetAuthor()]; }
  void visit(Comic &c) override { ++cat[c.getHero()]; }
};
```

- `se/visitor`:
  - cycle of includes
- `se/visitor2`:
  - Fix by forward declaring classes

Compilation Dependencies

- an include creates a dependency
- Best to reduce dependencies
  - Avoid cycles
  - Fewer recompilations
  - Faster compile time

Advice: *whenever possible*, forward declare a class

```
class XYZ; // Hey compiler, this type will exist
```

```
// File "a.h"
class A {...};
```

```
// File "b.h"
#include "a.h"
class B : public A {
    ...
};
```

```
// File "c.h"
#include "a.h"
class C {
  A a;
};
```

To construct C, you will need to know its size, and therefore need the size of A

```
// File "d.h"
class A;
class D {
  A *pA;
};
```

```
// File "d.cc"
#include "a.h"
```

```
pA->method();
```

```
// File "e.h"
class A;
class E {
    A foo(A a);
};
```

```
// File "e.cc"
#include "e.h"
#include "a.h"
A E::foo(A a) {}
```

Reducing Dependencies

```
// File "window.h"
#include <Xlib/Xlib.h>
class XWindow {
  Display *d;
  Window w;
  GC gc;
  public:
  drawRect();
  ...;
};
```

```
// File "client.cc"
#include "window.h"
```

`client.cc` needs to recompile even if a private member in `window.h` changes

Strategy: pImpl Idiom (pointer to Implementation)**

```
// File "windowimpl.h"
#include <Xlib/Xlib.h>
struct WindowImpl {
  Display *d;
  Window w;
  GC gc;
};
```

```
// File "window.h"
struct WindowImpl;
class XWindow {
    WindowImpl *pImpl;
    public:
    ...
};
```

```cpp
// File "window.cc"
#include "window.h"
#include "windowimpl.h"
XWindow() : pImpl { new WindowImpl } {}
~XWindow() { delete pImpl; }
```

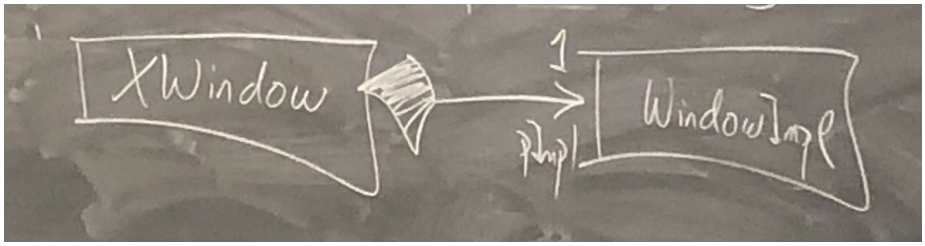


Figure 15: BRIDGE_1

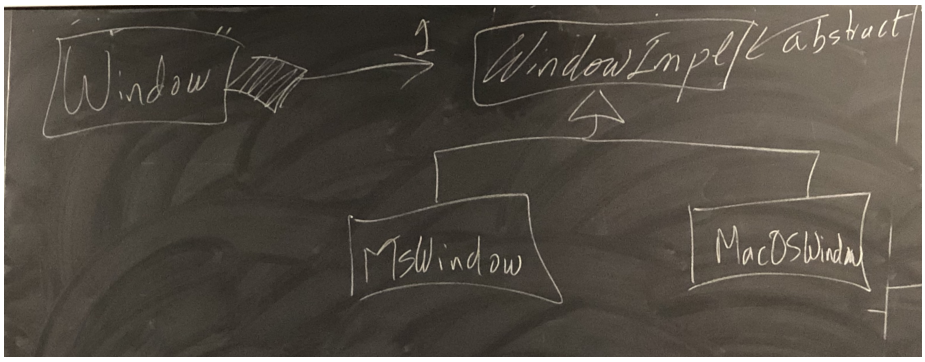


Figure 16: BRIDGE_2

Bridge Design Pattern

Extended the pImpl idiom to accomodate multiple implementation styles.

Lecture 22

MVC Design Pattern, Course Evals, Exception Safety (smart pointers)

Coupling & Cohesion

- **Coupling:** the interaction/dependency between modules

  - Aim -> low coupling
- **Cohesion:** how related are things within a module
  - Aim -> high cohesion

Decouple the Interface

- Primary program classes should not interact with the user - no I/O
- Single Responsibility Principle:
  - a class should be responsible for one task
  - a class should only have one reason to change

Model-View-Controller Design Pattern

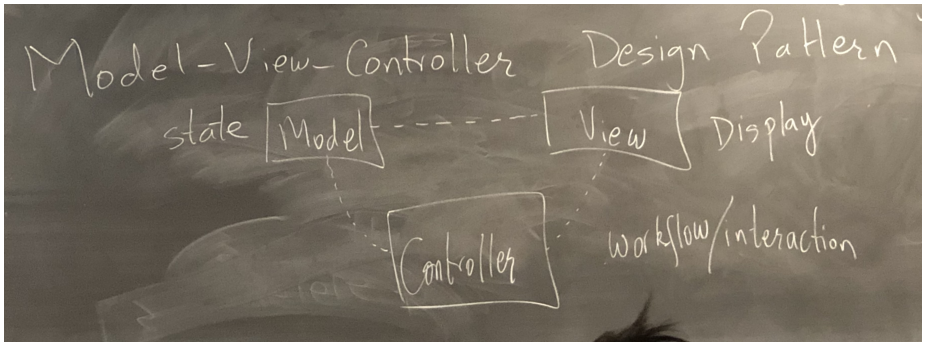


Figure 17: MVC_Pattern

Exception Safety

```
void f() {
    MyClass *p = new MyClass;
  MyClass mc;
  g(); // If g throws, f leaks memory
  delete p;
}
```

Even if exceptions occur, our code should:

- not leak memory
- not have dangling pointers
- not break class invariants
- continue to be in a “valid” state

```
void f() {
    MyClass *p = new MyClass;
```

```
    try {
        MyClass mc; // error-prone
        g(); // tedious
    } catch(...) {
        delete p;
        throw;
    }
    delete p;
}
```

C++ Guarantee

- During stack unwinding, destructors for stack allocated data are called.
- Maximize stack usage

RAII: Resource Acquisition Is Initialization

- wrap resources in stack objects whose destructor frees the resource
- ```
  ifstream f{"file.txt"};
  // file f will be closed irrespective of whether
  // an exception occurs or not.
  ```
- RAII with heap memory
  - wrap heap memory within a stack object
  - have the stack object's destructor delete the heap memory

Standard Library's template class `uniqueptr<T>`

- constructor takes a `T*` (pointer to heap memory)
- destructor calls delete on the provided pointer
- ```
  void f() {
    std::unique_ptr<MyClass> p{new MyClass};
    auto p = std::make_unique<MyClass>(); // Use for project
    MyClass mc;
    g();
  }

  unique_ptr<MyClass> p = make_unique<MyClass>();
  unique_ptr<MyClass> q {p}; // Calls copy constructor and Won't Compile
  ```
- Copy constructor/assignment operator are disabled to avoid double free error. However, move constructor/assignment operator is enabled.
- `shared_ptr<T>` uses reference counting to keep track of how many owners share this heap object
- destructor is smart

- decreases reference count
- deletes heap object when reference count goes to 0 # Lecture 23

Last time: smart pointers

`unique_ptr<T>`

- destructor deletes heap object

`shared_ptr<T>`

- reference counting
- copying increments refcount
- destructors decrement refcount
- delete if refcount is 0

Levels of Exception Safety

Basic Guarantee: If an exception occurs, the program is in a valid (memory intact), but unspecified state

Strong Guarantee: If an exception occurs while executing `f()`, it is as if `f()` was never called.

No-throw Guarantee: Function `f()` does not throw an exception and always achieves its goal.

```
class A{...};
class B{...};
class C{
  A a;
  B b;

  void foo() {
    a.method1(); // strong guarantee
    b.method2(); // strong guarantee
  }
};
```

Scenario: What if method2 fails?

- "undo" method1??
- typically impossible for non-local side-effects

Let's assume method1/method2 only have local side effects

- undoing method1 would mean undoing changes to object "a"

Idea: Work on copies of "a" and "b". If successful, swap with originals

```
void C::foo() {
    A tempA{a};
    B tempB{b};

    tempA.method1();
    tempB.method2();

    a = tempA;
    b = tempB; // if B::operator= throws, we lose our strong guarantees
}
```

Pointer assignment is no throw!!

```
struct CImpl{A a; B b;};
class C {
  unique_ptr<CImpl> pImpl;

  // Strong guarantee
  void foo() {
    unique_ptr<CImpl> temp = make_unique<CImpl>(*pImpl);
    // Also works: unique_ptr<CImpl> temp{new Cimpl{*pImpl}};
    temp->a.method1();
    temp->b.method2();
    swap(pImpl, temp);
  }
};
```

Exception Safety in the STL

`std::vector`

- uses RAII
- heap allocates array, destructor deallocates array

```
void foo() {
    vector<MyClass> v;
    ...
}
// When v goes out of scope, elements are automatically destroyed
// vector destructor frees the array
```

```
void g() {
    vector<MyClass *> v;
  for (auto &m : v) delete m;
}
```

```
void h() {
    vector<unique_ptr<MyClass>> v;
```

}

`vector::emplace_back` provides a strong guarantee

- Easy case:
  - size < cap
  - place object at end
  - increment size
- Harder case:
  - Less-efficient:
    - size == cap
    - create new larger array
    - copy from old to new (if exception, delete new array)
    - delete old (no-throw)
  - More-efficient:
    - create new larger array
    - move from old to new (if an exception occurs, you lose strong guarantee)
    - delete old

Note: only use move operations if they are labelled `noexcept`

Casting

Casting in C:

```
Node n;
int *p = (int *) &n;
*p = 10;
```

4 Casts in C++:

`static_cast`

```
void foo(int);
void foo(double);
...
double a = ...
foo(a); // foo(double)
foo(static_cast<int>(a));
```

```
Book *bp = new Text{...};
Text *tp = static_cast<Text *>(bp);
// unchecked cast
// Behaviour undefined if bp is not pointing to a Text
```

reinterpret_cast

- to treat any type as any other type

```
Student s;
Turtle *tp = reinterpret_cast<Turtle *>(&s);
tp->draw();
```

const_cast

- a cast to remove "const" from a variable

```
void foo(int * q) {...}
...
const int *q = ...;
foo(q); // won't compile
foo(const_cast<int *>(q)); // will compile
```

dynamic_cast

Fourth cast in next lecture...