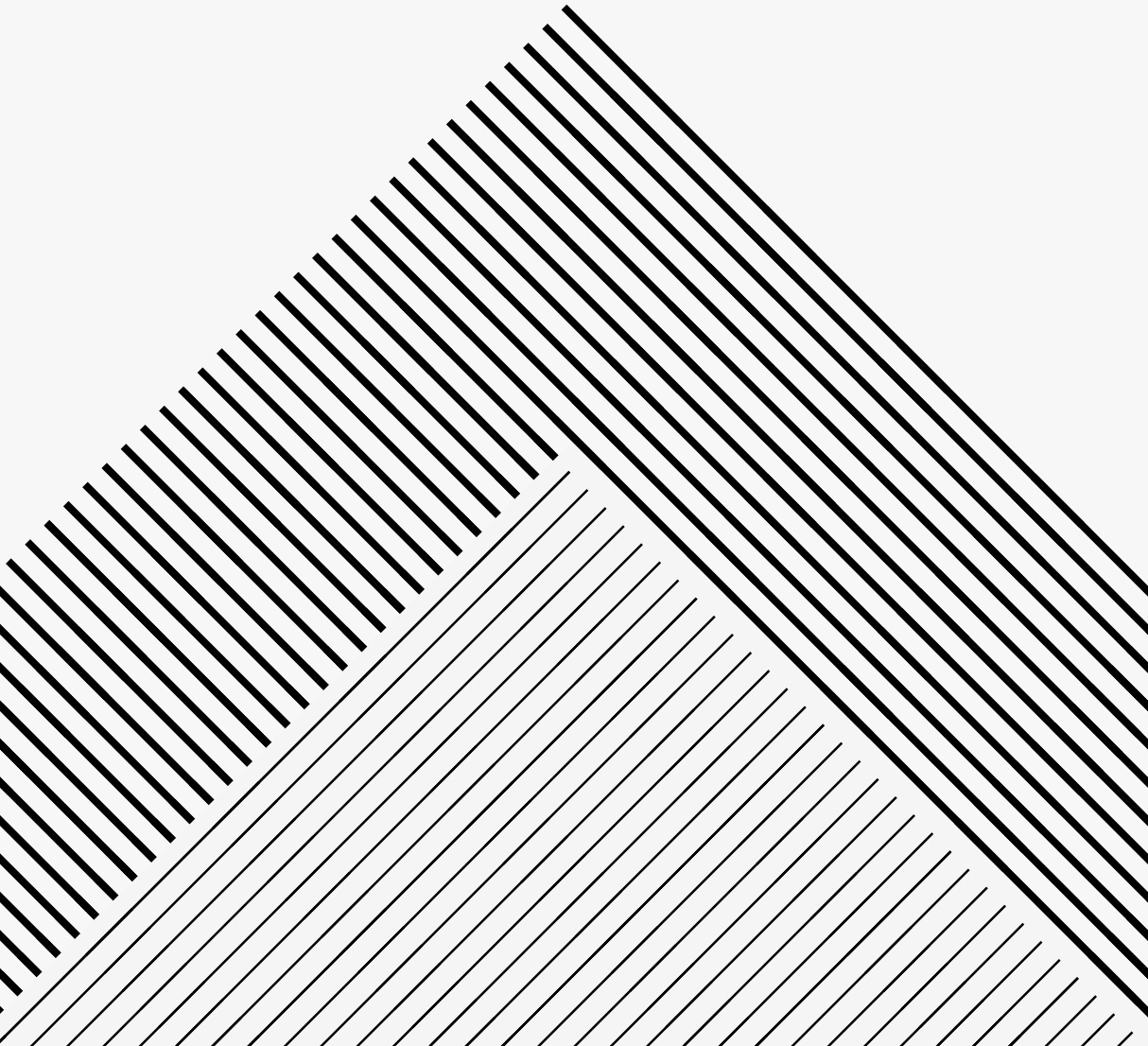


LECTURE 1

ADTS

- **ADT**: abstract data type
  - ↳ bundles tgt data w/operations & valid ranges that data can hold
  - ↳ allow us to hide complex details behind an **interface**
- client only needs to understand interface to use class & can ignore underlying implementation
  - ↳ e.g. in a BST, client can call insert/delete/search w/o needing to worry abt tree balancing
  - ↳ e.g in dictionary, can simply insert/lookup values w/o worrying abt internals, hash func being used, etc.
- ideally, use **compiler** to enforce certain properties of ADT
  - ↳ e.g. linked list ADT: 1 property we may wish to have is that all nodes are on the heap
- detecting issues at compile-time ensures that users of our software never encounter this particular suite of runtime bugs
- wish to prevent tampering & forgery of ADTs
  - ↳ **tampering**: modifying a valid instance of an ADT to become invalid in an unexpected way
  - ↳ **forgery**: creating invalid instance of ADT

- e.g. Rational ADT

```
#include <iostream>
using namespace std;
int main() {
   cout << "Enter a rational #" << endl;
   Rational r,p,q;  // default ctor
   cin >> r >> q;  // define what it means to read Rational from cin
   p = q+r;  // define what it means to add 2 Rationals
   cout << q/r << endl;  // define how to print Rationals
   Rational z {q};  // define copy ctor for Rationals
}
// Defining Rational class
class Rational {
   int num, denom;
public:
   Rational(): num{0}, denom{1} {}
   explicit Rational(int num): num{num}, denom{1} {}
   Rational(int num, int denom): num{num}, denom{denom} {}
};
```

- **explicit** keyword: used for single param ctors & prevents implicit conversions
  - ↳ e.g. `void f(Rational x) { ... }`
    - w/o explicit keyword, f(247) will compile b/c the Rational(int num) will be **invoked implicitly** & a Rational w/num = 247 & denom = 1 is created
    - w/explicit keyword, must instead write f(Rational{247})
  - ↳ helpful to catch mistakes
- implicit ctors are only invoked when there's a single argument so for multiple-argument ctors, explicit invocation is necessary
- e.g.

```
class Node {
   Node* next;
   int data;
public:
   Node(int data): data{data}, next{nullptr} {}
};
```

  - ↳ `q(Node n){ ... }` ⟶ q(4) would compile b/c Node{4} ctor is implicitly used

- e.g. void f(std::string s) {
  ... f("Hello World");
  }
  ↳ works b/c std::string has single-param ctor which takes in a char* & is non-explicit

- Rational ADT doesn't necessarily need to be a class
- ADTs aren't tied to one particular implementation
- classes are nicer than structs b/c:
  ↳ construction & destruction is guaranteed (on stack)
  ↳ can enforce legal value ranges via access specifiers (public / private / protected)
  ↳ e.g. w/ C struct, someone can easily set denom = 0 but w/ class, denom is private & can add logic to ctor to reject denom = 0
- invariant: smth abt class / ADT that's always expected to be true
- ADTs don't necessarily need a default ctor
  ↳ e.g. Student ADT w/ name, ID, & b-day fields has no sensible defaults
  ↳ if no ctors are specified, compiler provides default ctor
    - leaves primitive fields uninitialized
    - defaults constructs obj fields
  ↳ if we write any ctor ourselves, compiler-provided ctor is not supplied
- e.g. class Rational {
  int num, denom;
  public:
  Rational(int num = 0, int denom = 1): num{num}, denom{denom}{}
  };
  ↳ use default params for better code style so we only need 1 ctor
    - can create Rational objs w/ fewer than 2 arguments & default values will fill in missing ones
  ↳ default params must appear at trailing end of param list
    - Rational q; uses num = 0 & denom = 0
    - Rational w{1}; uses denom = 0
    - Rational z{1, 5}; uses no default vals
- e.g. importance of trailing default params:
  class Foo {
  public:
  Foo(string x = "hello", int y) ... // default param is not trailing, which is wrong
  Foo(int z) ...
  };
  ↳ calling Foo{5} would be ambiguous & wouldn't compile
- default params only appear in declaration, not definition, of func
  ↳ e.g.
  Rational.h:
  class Rational {
  int num, denom;
  public:
  Rational(int num = 0, int denom = 1)
  };
  Rational.cc:
  #include "Rational.h"
  Rational::Rational(int num, int denom): num{num}, denom{denom}{}
  ← defaults aren't provided

MIL

- use MIL (Member Initialization List) to set fields
  ↳ e.g. consider diff btwn:
  // using MIL to set fields
  Rational(int num, int denom): num{num}, denom{denom} {}

```
// using "this->" to set fields
Rational(int num, int denom) {
    this->num = num;
    this->denom = denom;
}
```

- in general, using MIL is better b/c it improves performance
  - ↳ consider how object is constructed:
    1) space is allocated on stack / heap
    2) call superclass ctor
    3) initialize fields via MIL
       - if not given value, field is default constructed
    4) run ctor body
  - ↳ using "this->" means fields are first given default values ? then replaced w/ assignment in ctor body
    - results in slower performance
  - ↳ can't use "this->" w/ const or reference member vars b/c they can't be assigned new values after initialization

LECTURE 2

MIL CONT

- using MIL is considered better style than assigning fields in ctor body
- there are cases where using MIL is **necessary**
  - ↳ **const fields**
    - e.g.
```
class Student {
    const int id;
  public:
    Student(int id) {
        this->id = id   // ← doesn't compile b/c const fields can't ever be changed
    }
};
```
  - ↳ **reference fields**
    - e.g.
```
class Student {
    University &myUni;
  public:
    Student(University &uni) {
        this->myUni = uni;   // ← doesn't compile b/c references must always be initialized & once initialized, they can't be reassigned to refer to diff obj
    }
};
```
  - ↳ **object fields w/o default ctors**
    - e.g.
```
Class A{
  public:
    A(int x){}
};

Class B{
  A myA;
  public:
    B() {
        myA = A{5};   // ← doesn't compile b/c A isn't initialized in MIL, but there's no default ctor for A
    };
```
  - ↳ **if superclass doesn't have default ctor**
    - e.g.
```
class A {
    int x;
  public:
    A(int x): x{x} {}
};

class B: public A{
    int y;
  public:
    B(int x, int y) {   // ← doesn't compile b/c superclass A has no default ctor & field x isn't provided in B's MIL
        this->x = x;
        this->y = y;
    }
};
```
    - → what if we gave A a default ctor?
```
class A {
    int x;
  public
    A(): x{0} {}
    A(int x): x{x} {}
};

class B: public A{
    int y;
  public:
    B(int x, int y) {
        this->x = x;   // ← doesn't compile b/c x is private in superclass A
        this->y = y;
    }
};
```
    - → correct way:
```
class B: public A {
    int y;
  public:
    B(int x, int y): A{x}, y{y} {}   // ← works even if A doesn't have default ctor
};
```
- in MIL, superclass is always initialized first & then member fields in order they're declared in

OVERLOADING

- **overloading** is implementing 2+ fcns/methods w/same name, but differing in #/types of arguments provided
  - ↳ e.g.
```
bool negate(bool b) {
    return !b;
}
```
```
int negate(int x) {
    return -x;
}
```
  - ↳ can't overload based solely on return type
- to perform **operator overload**, define fcn w/name "operator" concatenated w/operator symbol
  - ↳ e.g. operator+, operator>>, operator!, operator==
  - ↳ #arguments must match arity of operator
    - e.g. + is binary operator (2 args), ! is unary operator (1 arg)
- e.g. to support `cin >> r >> q;` where r & q are Rationals, define operator overload:

```
istream& operator>>(istream& in, Rational& r) {
    in >> r.num;
    in >> r.denom;
    return in;
}
```

(in: ↑ LHS of >>; r: ↑ RHS of >>)

  - ↳ istream is passed by ref b/c copying is disallowed for istreams
  - ↳ Rational is passed by ref b/c we want changes to be reflected outside this fcn
  - ↳ return in to allow for **chaining**
    - `cin >> r >> q` (`cin >> r` evaluated 1st)
    - if we return in, simplifies to `cin >> q`
  - ↳ problem: fcn is implemented outside of class, but r.num & r.denom are private fields & can't be accessed in operator>>
    - sol 1: provide **accessor methods**
      - → getNum & getDenom return refs to num & denom fields
      - → sometimes paired w/ **mutator methods** setNum & setDenom (could enforce invariants like denom ≠ 0 w/these methods)
      - → sometimes called getters/setters
    - sol 2: declare operator>> as **friend fcn** (better option)
      - →
```
class Rational {
    int num, denom;
    ...
    //can be in any section (private or public)
    friend istream& operator>>(istream& in, Rational& r); ← fcn can access any private fields & methods of Rational
}
```
- e.g. support `p = q + r`, where p, q, & r are Rationals

```
Rational operator+(const Rational& lhs, const Rational& rhs) {
    return Rational{lhs.num * rhs.denom + lhs.denom * rhs.num, lhs.denom * rhs.denom};
}
```

  - ↳ would need to be a friend as well
  - ↳ take in args via constant ref
    - constant b/c don't want lhs/rhs to change
    - ref b/c quick & no copying
- declaring all operator overloads as friends is pain
  - ↳ alt: define operator overloads as methods in Rational
    -
```
class Rational {
    int num, denom;
  public:
    ...
    Rational operator+(const Rational& rhs) {
        return Rational{num * rhs.denom + denom * rhs.num, denom * rhs.denom};
    }
};
```

→ this takes place of lhs
→ Rational obj can all access each other's private fields
- no friend is necessary b/c fcn is defined in class

- r+q ≡ r.operator+(q);
  ↳ ≡ means semantically equivalent
- operator<< & operator>> are usually defined as standalone fcns b/c cin/cout appear on LHS
- e.g. support `r+5` where r is Rational

```
class Rational {
  ...
  Rational operator+(int rhs) {
    ...
  }
};
```

- e.g. support `5+r` is diff b/c order of args matters
  ↳ want int on LHS so we need standalone fcn:

```
Rational operator+(int lhs, const Rational& rhs) {
  return rhs+lhs;
}
```

- e.g. support `p=q+r`, where we're setting 1 Rational to another
  ↳ compiler provides `copy asgnt operator`, but can also write own
  ↳ class Rational {

```
  .
public:
  Rational& operator=(const Rational& rhs) {
    num = rhs.num;
    denom = rhs.denom;
    return *this;
  }
};
```

- return type is Rational& b/c we can also chain operator=
  → e.g. a=b=c=d=e;
  → evaluates right to left & returns value that was set

LECTURE 3

OPERATOR OVERLOADING CONT

- there are some operator overloads that must be defined as methods (i.e. can't be standalone fcns):
  - ↳ operator =
  - ↳ operator [ ]
  - ↳ operator →
  - ↳ operator ( )
    - e.g.

```
class A {
  :
public myA{1,2,3};
};
```

```
myA();
  ↑
functor, which means obj
is treated as fcn
```

  - ↳ operator T
    - T is type
- e.g support division of Rationals `q/r`

```
class Rational {
   int num, denom;
public:
   ...
   Rational operator / (const Rational& rhs) {
      return Rational{num * rhs.denom, denom * rhs.num};
   }
};
```

- e.g. support `cout << r << endl` where r is Rational
  - ↳ must be standalone fcn b/c LHS is ostream

```
class Rational{
   ...
   friend ostream& operator << (ostream& out, const Rational& r);
};
ostream& operator<<(ostream& out, const Rational& r){
   out << r.num << "/" << r.denom;
   return out;
}
```

  - ↳ usually don't provide endl in operator<< defn b/c don't want to force users to use new lines
  - ↳ consider:

```
ofstream{"file.txt"};
Rational r{5,7};
file << r << endl;
```

    - supported by prev defined method b/c param ostream is general type
- operator<< & operator>> defns also work for reading from & printing to files (& other types of streams)
- Rational z{q}; or Rational z=q; or Rational z(q);
  - ↳ running `copy ctor`
    - diff from copy asgnt operator b/c an obj is created for the 1st time (doesn't exist before)
  - ↳ compiler provides copy ctor that simply copies all fields, but can also write own:

```
class Rational {
  ..
public:
  Rational(const Rational& other): num{other.num}, denom{other.denom}{}
};
```

- having written all of these operator overloads, Rational ADT is easy to use from client perspective
- overloading operators gives us convenient syntax

- classes allow us to enforce invariants
  ↳ good ADT design
- friends can be used sparingly to give access to private fields

VALUE CATEGORIES

- every expression in C++ has both type & value category
- we discuss 2 categories:
  ↳ **lvalue** is any expr you can take address of
    - e.g.
      ```
      int x = 5;
      f(x);
      ```
      can take address of expr "x" so it's an lvalue
  ↳ **rvalue** is temp value so it will be destroyed "soon"
    - e.g
      ```
      f(5);
      ```
      can't take address of expr "5" so it's an rvalue
- e.g.
  ```
  string f() {
    return "Hello World";
  }
  string s = f();
  ```
  expr is rvalue b/c returned result of f only exists until end of line
  ↳ can't run &f() b/c string isn't stored anywhere permanently, just put in a temp place until it's saved into s

- references we've seen so far are **lvalue references**
  ↳ can only bind to exprs that are lvalues
  ↳ e.g.
  ```
  int x = 5;
  int &y = x; ✓ "x" is lvalue
  // However:
  int &z = 5; X doesn't compile b/c 5 is rvalue
  ```
  ↳ exception is binding rvalues to **const lvalue refs**
    - e.g.

`void f(int& x){`	`void g(const int& x){`
`...`	`...`
`}`	`}`
`f(5)` X	`g(5)` ✓ allowed b/c we won't modify x & compiler creates temp mem location to store 5 in

- can also create **rvalue refs**, which extend lifetime of rvalue to lifetime of ref
  ↳ e.g.
  ```
  string f() {
    return "Hello World";
  }
  string&& s = f();
  ```
  syntax for rvalue ref
    - can use temp value returned by f for as long as s exists
- most commonly, rvalue refs are used for overloading fcns based on value category of expr
  ↳ e.g.
  ```
  void f(const string& s) {
    cout << "1" << endl;
  }
  void f(string&& s){
    cout << "2" << endl;
  }
  string s{"CS247"};
  ```

```
f(s);  // 1 is outputted
f(string{"CS247"}); // 2 is outputted
```

- type + value categories are indep categories
  - ↳ e.g.

```
void f(string &&s) {
  cout << s << endl;
}
```

  - although s references an rvalue, we can take its address so its value category is lvalue

LINKED LIST ADT

- leverage lvalue / rvalue knowlegde for efficiency
- focus on invariants + encapsulation later, so only want correctness + efficiency now
- e.g.

```
Struct Node {
  string data;
  Node* next;
};
```

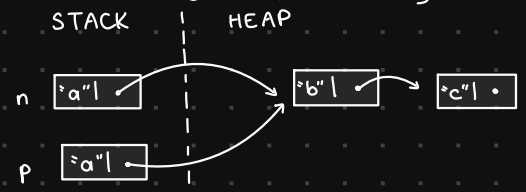
  - ↳ we want client code:

```
Node n{"a", new Node{"b", new Node{"c", nullptr}}};
Node p{n};
p.next->data = "z";
cout << p;  // a, z, c
cout << n;  // a, b, c
```

- ctor:

```
Node::Node(const string& data, Node* next): data{data}, next{next} {}
```

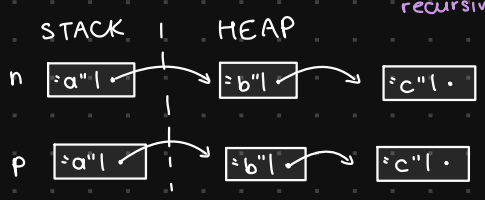
- support `Node p{n};`
  - ↳ executes copy ctor provided by compiler, which copies each field



  - p.next->data = "z" modifies n as well
  - we have shared data
  - ↳ define own copy ctor that recursively copies data structure:

```
Node::Node(const Node& other): data{other.data},
  next{other.next ? new Node{*(other.next)} : nullptr} {}
```

recursively calls copy ctor



  - this is deep copy, as opposed to a shallow copy

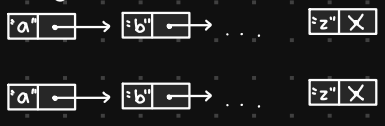
LECTURE 4

LINKED LIST ADT CONT

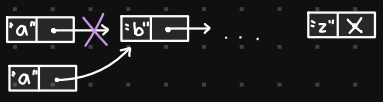
- e.g.
```
{
  Node n{"a", new Node{"b", new Node{"c", nullptr}}};
}
```
↳ issue is that although n is freed automatically, rest of Nodes in list are on heap & must also be freed

- write **dtor** that runs when stack allocated objs go out of scope & when heap allocated objs are freed
↳ `Node::~Node(){ delete next; }`

- e.g.
```
Node getAlphabet(){
  return Node{"a", new Node{"b", ..., new Node{"z", nullptr}}...};
}
Node n{getAlphabet()};
```
↳ looks like this in mem:



↳ copied 26 nodes, then deleted originals immediately (inefficient)
↳ getAlphabet() is **rvalue** so it's going to be deleted by next line
- no one will use that mem so we can steal it
- write **move ctor** for rvalues, which is faster:
```
Node::Node(Node&& other): data{other.data}, next{other.next}{
  other.next = nullptr;
}
```
→ in mem:



↳ 1 more slight optimization for `data{other.data}` is that other.data is lvalue b/c it has an address so code invokes copy ctor for data str
- since other.data will die soon, we prefer to run str's move ctor:
`data{std::move(other.data)}`
→ **std::move** is included in <utility> & forces lvalue to be treated like rvalue

- w/ default copy asgnt operator, only shallow copies of pointers are made
↳ e.g.
```
Node n{"a", new Node{"b", new Node{"c", nullptr}}};
Node p{"d", new Node{"e", nullptr}};
p = n;  // runs copy asgnt operator
```
- results in **data sharing** & leaked Nodes

- to implement our own **copy asgn t operator**:
↳ attempt 1:
```
Node& Node::operator=(const Node& other){
  delete next;
  data = other.data;
  next = other.next ? new Node{*(other.next)} : nullptr;
  return *this;
}
```
- example of breaking:
```
Node n{"a", new Node{"b", new Node{"c", nullptr}}};
n = n;
```
→ in mem:

- to protect against self-asgnt, add this to beginning of asgnt operator:
  if (this == &other) return *this;
- new has possibility to reject request for more mem so we quit fcn in that case
  → call new first so if it fails, we haven't deleted anything yet

↳ attempt 2:

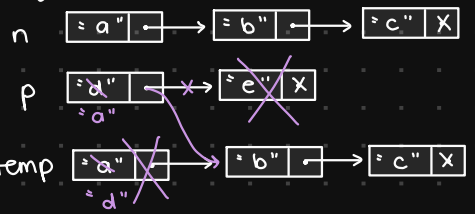
```
Node& Node::operator=(const Node& other) {
    if (this == &other) return this;
    Node* temp = other.next ? new Node{*(other.next)} : nullptr;
    data = other.data;
    delete next;
    next = temp;
    return *this;
}
```

- there's some code duplication btwn copy ctor & copy asgnt operator so we use **copy & swap idiom**

↳ attempt 3 (final):

```
Node& Node::operator=(const Node& other) {
    Node temp{other};
    std::swap(data, temp.data);
    std::swap(next, temp.next);
    return *this;
}
// temp reaches end of scope & is destructed
```

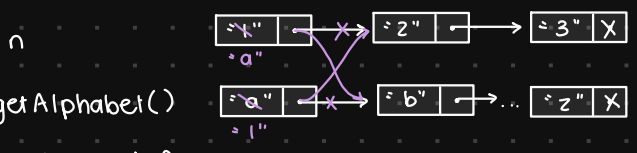
- **swap** is in <utility> & swaps 2 vals
- e.g. p = n;



- implement **move asgnt operator** for more efficiency:

```
Node& Node::operator=(Node&& other);
    std::swap(data, other.data);
    std::swap(next, other.next);
    return *this;
}
```

↳ e.g. n = getAlphabet();



- if we don't define move ctor / asgnt op, but do define copy ctor / asgnt op, compiler will use copies in all scenarios
- **Rule of 5 / Big 5**: if one of following is written, all should be written — dtor, move ctor, move asgnt operator, copy ctor, copy asgnt operator

ELISION

- e.g.

```
Rational makeRational() { return Rational{1, 5}; }
{
    Rational r = makeRational();   // what runs here?
}
```

↳ neither move nor copy ctor is called due to elision
- in certain cases, compiler skips calling copy/move ctors & instead writes obj's vals directly into its final location
- e.g

```
void doMath(Rational r){
  ...
}
doMath(makeRational()); // {1,5} is written directly into r
```

- elision is possible even if it changes program output

disable w/ flag: `-fno-elide-constructors`

↳ slows down program

LECTURE 5

IMPROVING ENCAPSULATION

- e.g. how client can misuse ADT:

```
struct Node {
    Node * next;
    string data;
};
```

↳ e.g forgery

```
Node n{"bad ptr", (Node *) 0x247};
```

completely random mem address

  - most likely seg fault on any kind of use

↳ e.g. tampering

```
Node n{"a", nullptr};
n.next = &n;   // cycle of 1 node
```

↳ e.g. tampering

```
Node n {"abc", nullptr};
Node p {"def", &n}; // tries to delete stack mem when dtor runs
```

- every ADT has rep in mem
  ↳ e.g. linked lists are implemented via series of nodes
- all above issues form a problem called **representation exposure** (i.e. client gains access to mem)
  ↳ client can use this knowledge to do unexpected things
- we have expectations/rules for all linked lists:
  ↳ all next nodes are heap allocated (or next is nullptr)
  ↳ no sharing btwn separate lists
  ↳ no cycles in list
- clients shouldn't need to uphold **invariants**, it's our responsibility
  ↳ soln is **encapsulation**: provide public methods that clients can use to interact w/ ADT
    - e.g.

```
// List.h
class List {
    struct Node;          fwd declaration (i.e. don't need to provide implementation) of private, nested class
    Node * head = nullptr;     front of list
public:
    List();          creates empty List
   ~List();
    void push(const string& s);
    string& ith(int i);
};
// List.cc
#include "List.h"
struct List::Node {
    Node * next;
    string data;
};
List::List() {}
List::~List() { delete head; }
void List::push(const string& s) {
    head = new Node{s, head};
}
string& List::ith(int i) {
    Node * cur = head;
    for (int x = 0; x < i; ++x)
```

```
    cur = cur->next;
  return cur->data;
}
```

→ solves rep exposure problem b/c client has no knowledge of nodes & no access to internal mem rep so no tampering / forgery can be done

→ soln makes Lists slow b/c ith is not constant time operation (takes i steps to find i^{th} node in List):

```
for (i=0; i<10000; ++i) cout << l.ith(i);
```

Total: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$= \frac{n^2}{2} + \frac{n}{2}$

$\in O(n^2)$

ITERATORS

- **design patterns**: effective slns to common problems
- problem is efficient looping over data structure while maintaining encapsulation
  - ↳ design pattern to solve this is **iterator pattern** (provides abstraction for pointers)
    - e.g.

```
//example of pointer to loop thru arr
int a[5] = {247, 1, -100, 5, 605};
for (int* p = a; p != (a+5); ++p) {
   cout << *p << endl;
}
```

- e.g.

```
// clients use iterator as such (note similarities btwn pointer & iterator):
List l;
for (List::Iterator it = l.begin(); it != l.end(); ++it) {
   cout << *it << endl;
}
```

→ for List, need to implement begin & end fcns which return iterators

→ for List::Iterator, need to implement fcns:

1) != to compare
2) ++ to go to next Node
3) * to get string at current node

- e.g. implementation of above fcns

```
class List {
  ...
 public:
   class Iterator {
     Node* cur;
    public:
      Iterator(Node* cur): cur{cur} {}
      bool operator!=(const Iterator& o){
          return cur != o.cur
      }
      string& operator*(){
         return cur->data;
      }
      Iterator& operator++(){
         cur = cur->next;
         return *this;
      }
   }; //end of Iterator class
   Iterator begin() {return Iterator{heads};}
   Iterator end() {return Iterator{nullptr};}
```

```
};
```

- if iterator is implemented like above, can use **range-based for loop syntax**
  - ↳ e.g. `for (string s : l) cout << s << endl;`
    - no ref always implies copy from cur→data into s
- e.g.

```
// no effect on list
for (string s : l) {
  s = "";
}
// sets strings in list to empty
for (string &s : l) {
  s = "";
}
```

- can also use **auto** for type deduction
  - ↳ e.g. `for (auto s : l)` ← determines s is string for us
  - ↳ auto **drops refs** so if we want to modify l, use auto&
- another problem is that Iterator is public nested class w/ public ctor
  - ↳ worry of forgery
    - e.g. `auto it = List::Iterator{nullptr};`
      - → could be problem for other kinds of iterators where nullptr is not used
  - ↳ soln is give Iterator a **private ctor**:

```
class List {
  ...
  public:
    class Iterator {
      Node *cur;
      Iterator(Node *cur) : cur{cur} {}
    public:
      // as before
      friend List; // Lists can use private methods of Iterators, namely ctor for begin & end fcns
    };
  ...
};
```

- in general, be cautious when using friends b/c it weakens encapsulation
- e.g. subtle issue:

```
void useList(const List& l) {
  ...
  for (string s : l) {...}
}
```

  - ↳ doesn't compile b/c compiler doesn't know if l.begin() & l.end() that it won't modify fields of const List& l
  - ↳ soln is to declare begin & end as const methods
    - `Iterator begin() const {return Iterator{head};}`
    - `Iterator end() const {return Iterator{nullptr};}`
- constant obj (or const ref to an obj) may only have const methods called on it

LECTURE 6

PHYSICAL VS LOGICAL CONSTNESS

- can modify a const List& by using its iterator due to diff btwn physical & logical constness
- **physical constness** is abt particular obj's implementation
  - ↳ compiler guarantees this for const objs/refs
  - ↳ ensures fields won't change
- **logical constness** asks not only for fields to be immutable, but also that mem associated w/ ADT doesn't change
  - ↳ to achieve this:
    - begin & end can't be const methods b/c they return iterators that return modifiable string refs via operator*
    - write additional methods **cbegin & cend**, which return ConstIterators, which provide const string& when calling operator*
    - class List {

```
...
public:
    class ConstIterator {
      Node* cur;
       ConstIterator(Node* cur): cur{cur} {}
    public:
       const string& operator*() {return cur->data;}
        bool operator!=(const ConstIterator& o) {...}
        ConstIterator& operator++() {...}
    };
   Iterator begin(){return Iterator{head};}
   ConstIterator cbegin() const {return ConstIterator{head};}
   Iterator end() {return Iterator{nullptr};}
   ConstIterator cend() const {return ConstIterator{nullptr};}
}
```

  - → now, can't call begin & end on const List&
  - → can only call cbegin & cend, which don't allow us to modify list
- above soln has **duplication** btwn 2 iterators
  - ↳ fix w/ **template class**, which is allowed to be parametrized via type provided
    - e.g. STL (std template lib):
      vector<int> vs vector<string>
      (type param denotes what vector contains)
  - ↳ specify return type of operator* as **template param** & rest of code is same
    - class List {

```
...
public:
   template <typename T> class MyIterator {
       Node* cur;
        MyIterator<T>(Node* cur): cur{cur} {}
    public:
       T operator*() {return cur->data;}
        bool operator!=(const MyIterator<T>& other) {...}
        MyIterator<T>& operator++() {...}
        friend List;
   }; // back in List
   using Iterator = MyIterator<string&>;
   using ConstIterator = MyIterator<const string&>;
   Iterator begin() {return Iterator{head};}
   ConstIterator cbegin() const {return ConstIterator{head};}
```

(} type aliases)

```
    // end & cend are same
};
```

- note that template defn of methods must be written in header file
  - ↳ Node's defn must also be moved into header file b/c template needs to know next field exists
- for each use of MyIterator<T>, compiler generates another copy of class w/ T sub where necessary
  - ↳ after, compiles as usual
- there's no run-time cost using templates, it's as if we wrote our own Iterator/ConstIterator

SEPARATE COMPILATION

- in List.h (og ver), provided fwd declaration of struct Node
  - ↳ provided defn of class in .cc file
  - ↳ esp useful for **separate compilation**: `g++ -std=c++14 List.cc -c`
    - -c compiler flag lets us prod obj file
    - think of obj files as containing machine ins for that code
    - use obj files to store result of compiling & reuse it if .cc files haven't changed
    - w/ many files, significant speedup while developing
- prefer to put **implementations in .cc file** for following reasons:
  - ↳ List.cc changes
    - recompile List.cc into List.O
    - link obj files tgt to get an executable
  - ↳ List.h changes
    - all .cc files that include List.h must recompile
    - if A.h includes List.h, any cc files including A.h must change
    - relink all .o files tgt
- prefer fwd declarations of classes where possible to minimize recompilation

- e.g. when can we simply fwd declare & when do we need to use #include?

```
//A.h
class A {...};
//B.h
class B : public A {
  ...
};
//C.h
class C {
   A myA;
};
//D.h
class D {
  A* aP;
};
//E.h
class E {
   A f(A x);
};
//F.h
class F {
  A f(A x) {
     x.doMethod();
     ...
  }
};
```

  - ↳ B & C require #include in order to determine their size & compile them
  - ↳ in D, all ptrs are same size so fwd declaration suffices

↳ in E, don't need to know size of A? just that A exists for type-checking purposes
  - fwd declaration
↳ in F, #include "A.h" to know doMethod exists

LECTURE 7

PREPROCESSOR DIRECTIVES

- #ifndef pattern is preprocessor directives, which are commands that allow transformations of source code before it's compiled
  - ↳ e.g.
```
#ifndef MOTION2D_H
#define MOTION2D_H
...
#endif
```
  - ↳ #include "file" replaces line w/ contents of file
- e.g. linear algebra modules
```
// Vec.h
struct Vec {
    int x, y;
};
//Matrix.h
#include "Vec.h"
struct Matrix {
    Vec e1;
    Vec e2;
};
//main.cc
#include "Vec.h"
#include "Matrix.h"
int main() {
    Vec v1 {1, 2};   // fields are public in struct so don't need explicit ctor
    Vec v2 {2, 3};
    Vec v3 = v1 + v2;
    Matrix m {v1, v2};
};
```
  - ↳ main.cc includes Vec.h & Matrix.h (which includes Vec.h)
    - 2 defns of struct Vec in main.cc, which is not allowed
- to fix issue of multiple defns, use include guards
  - ↳ e.g. #ifndef, #define, #endif
  - ↳ `#ifndef VARIABLE`
    ...
    `#endif`
    - stands for if not defined, then define
    - code btwn #ifndef & #endif will only be seen by compiler if VARIABLE is not defined
  - ↳ `#define VARIABLE` defines preprocessor var
  - ↳ e.g.
```
// Vec.h
#ifndef VEC_H
#define VEC_H
struct Vec {
    int x, y;
};
#endif
//Matrix.h
#ifndef MATRIX_H
#define MATRIX_H
#include "Vec.h"
```

```
struct Matrix {
  Vec e1;
  Vec e2;
};
#endif
```

- include guard works b/c once File.h is included once, var becomes defined so all future uses of include will omit block of code btwn #ifndef / #endif
  - ↳ issue w/ circular dependencies
    - e.g.

```
// A.h
#ifndef A_H
#define A_H
#include "B.h"
class A {
  B* myB;
};
#endif
// B.h
#ifndef B_H
#define B_H
#include "A.h"
class B {
  A* myA;
};
#endif
```

→ each class needs to know other exists before it's created
→ soln is to use fwd declarations

SEPARATE COMPILATION

- speeds up compilation / development time
  - ↳ change 1 .cc file, then update .o file / relink w/other old .o files
- if we change .h file, then many .cc files might need to be recompiled
  - ↳ might be faster to recompile everything instead of figuring out dependencies / only recompiling relevant .cc files
  - ↳ soln is to use build automation system
    - keeps track of what files have changed
    - keeps track of dependencies in compilation
    - just recompiles minimal # files to make new executable
- use make as build automation system
  - ↳ e.g create Makefile

main.cc , List.h , List.cc
(main.cc ↑ includes List.h; List.cc ↑ includes List.h)

- Makefile (VI):

```
myprogram: main.o List.o
(TAB) g++ main.o List.o -o myprogram
main.o: main.cc List.h
    g++ -std=c++14 main.cc -c
List.o: List.cc List.h
    g++ -std=c++14 List.cc -c
```

(myprogram: target; main.o List.o: dependencies; g++ ... myprogram: recipe; (C++ ver doesn't matter when linking))

  - ↳ typing "make" creates 1st target
    - looks at last modified time of dependencies

- if last modified time is newer for a dependency than a target, then that means target is out of date so recreate it

↳ V1 is still too much work b/c it requires lot of updates to Makefile

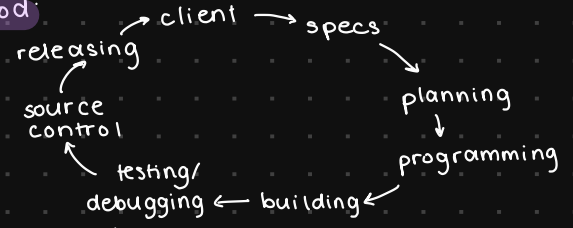
- **Makefile V2**:

```
CXX = g++   // special Makefile var for compiler we're using
CXXFLAGS = -std=c++14 -Wall -g -MMD → supports -include ${DEPENDS}
                          ↓ more warnings   ↘ debugging
EXEC = myprogram
CCFILES = $(wildcard *.cc)
OBJECTS = ${CCFILES:.cc=.o}
DEPENDS = ${CCFILES:.cc=.d}
${EXEC}: ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}
-include ${DEPENDS}    //compiles all obj files w/dependencies using CXX & CXXFILES
```

LECTURE 8

METHODOLOGIES

- software development lifecycle is process used in software dev to design, develop, test, & maintain software systems
- common development methodologies:
  - ↳ **waterfall method**: clients → specs → program → build → test / debug it → use source control → release it
  - ↳ **agile method**:



    - client gets looped into process
    - typically, work is done in 1-2 week "sprints"
- waterfall method is rarely used

TESTING AND DEBUGGING

- use mainly valgrind and GDB for **debugging** in C++
- **valgrind**'s primary use is **mem checking** tool
  - ↳ provides virtual CPU that it executes program on
    - checks for mem access, etc.
  - ↳ typically programs executed thru valgrind are 3-4x slower than usual
    - don't use for performance monitoring
- important when using valgrind / GDB, compile w/ **-g flag**
- using **--leak-check=full** allows us to see where leaked mem was allocated
- valgrind can only inform us of mem leaks in one particular invocation of program
  - ↳ important to design various test cases that hit each part of program
- valgrind can also detect **invalid reads**
  - ↳ reports line of source code where it happened
- **invalid write** gives 3 pieces of info:
  - ↳ where it occurred
  - ↳ where mem was freed
  - ↳ where mem was initially allocated

  valgrind can also detect invalid uses of delete / delete[]

- **GDB** stands for GNU debugger
  - ↳ useful for inspecting vars & seeing control flow of program
  - ↳ cmd is **gdb ./myprogram**
- various ways to run program
  - ↳ run: runs program
  - ↳ run arg1 arg2 arg3
  - ↳ run < test7.in
- setting a **breakpoint**:
  - ↳ break file: line #
  - ↳ break fcn-name
    - sets a breakpoint & stops program atp
- next / n: runs 1 line of program
- layout src: nicer display for source code
- print var: allows us to see val of that var atp
- refresh: fixes layout src display
- step: runs 1 ins of program
- list: shows surrounding lines of program

backtrace : lists seq of fcn calls that got us to current location

- up/down : change fcn in call stack we're observing so we can inspect other vars
- set var : sets var to smth else at runtime
- continue : runs program until next breakpoint
- display var : prints val of var after each next/step
- undisplay 1 : stops displaying 1st var set w/ display
  ↳ can't use var name ∴ must use # associated with it
- watch var : breaks program whenever var is changed
- delete breakpoint # : remove breakpoint from use (watch ∴ break)

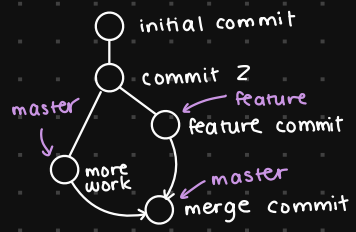
LECTURE 9

SOURCE CONTROL

- source control allows for management of source code for program
  - ↳ we want to collaborate w/others & have version history
  - ↳ historically, ppl have used SVN (Subversion) & Mercurial
- most ppl use git nowadays
- to create git repo, git init from within directory
- git has staging area, which allows us to prep which parts to be committed
- a commit is a unit of work
  - ↳ commit early & often
- git add: adds files to staging area
- git status: lets us see which branch we're on, which files/changes will be committed, etc.
- git add -p: lets us stage interactively
- git diff: see diff btwn unstaged files & most recent commit
- git diff --staged: see diff btwn staged files & most recent commit
- commit history looks like:

- may want to develop features or work on refactors separately from other dev
  - ↳ git branch branchName
    - makes new branch
  - ↳ git checkout branchName
    - switch btwn branches
  - ↳ e.g.

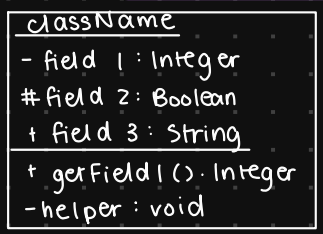
```
git branch feature
git checkout feature
git commit
git checkout master
git commit
git merge feature
```



  - ↳ git merge branchName
    - creates new commit w/parents that are latest commits on current branch & branchName
- git log: shows history of commits
- git push: send work to remote branch
- git pull: pulls work from remote branch & merges as well

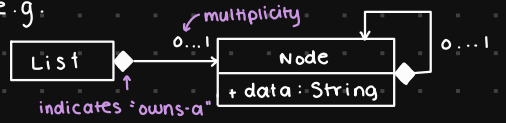
SYSTEM MODELLING

- visualize classes & rltnships btwn them at high lvl
- popular std is Unified Modelling Language (UML)
  - ↳ e.g.

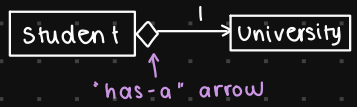


    - - : private
    - # : protected
    - + : public

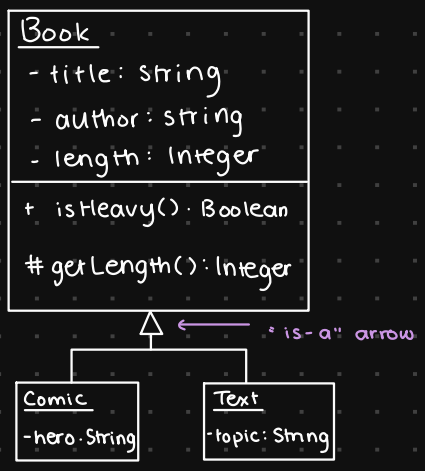
- modelling class rltnships:
  - ↳ **composition**: "owns-a" rltnship
    - if A "owns-a" B, generally B does not have independent existence from A
      - → if A dies, then B dies
      - → if A is copied, then B is copied as well (deep copy)
    - typically, implemented in C++ via obj fields
    - e.g.



  - ↳ **aggregation**: "has-a" rltnship
    - if A "has-a" B, generally B has independent existence outside of A
      - → if A dies, B lives on
      - → if A is copied, B isn't (shallow copy)
    - generally implemented via references or non-owning pointers (ones we don't delete)
    - e.g. Quest-like system



      - → if a Student dies, University lives on
      - → if we copy Student, don't copy University
  - ↳ **specialization**: "is-a" rltnship
    - 1 class is a version of another class
    - typically subclassing or subtyping rltnship
    - if B "is-a" A, then we should be able to sub smth of type B, wherever we expect smth of type A
    - in C++, achieve this via public inheritance
    - e.g.




```
Class Book {
    string title, author;
    int length;
  protected:
    int getLength() const {
        return length;
    }
  public:
    Book(string title, string author, int length): title{title}, author{author}, length{length} {}
    virtual bool isHeavy() const {
        return length > 200;
    }
};
```

```
Class Text : public Book {
  string topic;
 public:
  Text(string title, string author, int length, string topic): Book{title, author, length},
   topic{topic}{}
  bool isHeavy() const override {
    return getLength() > 500;
  }
};
```

→ why `Book{title, author, length}` in Text ctor?
→ recall 4 steps of obj construction
1) space is allocated on stack/heap
2) call superclass ctor ← runs this
3) initialize fields via MIL
4) run ctor body
→ can't set title, author, & length b/c they're private to Book
→ if we don't specify how to initialize Book portion, it will use Book's default ctor (won't compile if it doesn't exist)

- **virtual** keyword is used for achieving dynamic dispatch for polymorphic types
  ↳ **dynamic dispatch** means to choose which fcn to use at run-time rather than compile-time
  ↳ **polymorphic types** means 1 type that can take on multiple diff obj types
- e.g.

```
Book *b;
string choice;
cin >> choice;
if (choice == "Book")
    b = new Book{ ... };
else
    b = new Text{ ... };
cout << b->isHeavy() << endl;
```

↳ b has 2 types associated w/it
- static type: b is Book* ? compiler knows this
- dynamic type: describes what type b is pointing to at run-time
  → may depend on user input ? can't be determined ahead of time by compiler

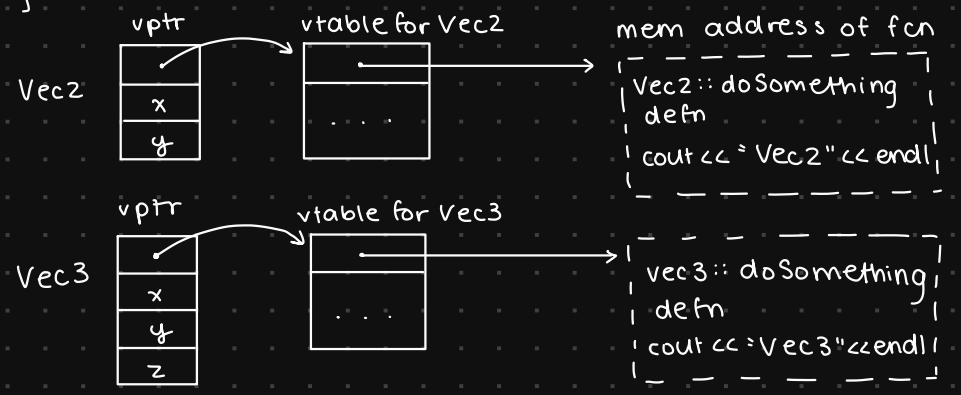
LECTURE 10

VIRTUAL, OVERRIDE, AND PURE VIRTUAL

- how do we know which method call in hierarchy is invoked for b.isHeavy() or b→isHeavy()?
  1) is method being called on an obj?
     - always use static type to determine which method is called
       → e.g.
       ```
       Book b {...}
       Text t {...}
       b.isHeavy()  // calls Book::isHeavy
       t.isHeavy()  // calls Text::isHeavy
       Book b2 = Text {...}
       b2.isHeavy() // calls Book::isHeavy
       ```
  2) is method called via pointer or ref?
     - if it's not declared as virtual, use static type to determine which method is called
       → e.g.
       ```
       Book *b = new Text {...};
       b→nonVirtual(); // calls Book::nonVirtual
       ```
     - if it's virtual, use dynamic type to determine which method is called
       → e.g.
       ```
       Book *b = new Text{...};
       b→isHeavy(); // calls Text::isHeavy
       ```
- e.g. can support:
```
vector<Book*> bookcase;
bookcase.push_back(new Book{...});
bookcase.push_back(new Text{...});
bookcase.push_back(new Comic{...});
for (auto book: bookcase){
    cout << book→isHeavy() << endl;
}
```
↳ each iteration calls a diff isHeavy method
- e.g. what abt (*book).isHeavy()?
  ↳ calls correct ver of isHeavy b/c *book yields a Book&
- override keyword has no effect on executable that's created
  ↳ helpful for catching some bugs
  ↳ e.g.
```
class Text {
  ...
  bool isHeavy();
};
```
missing a const so it won't override Book's isHeavy b/c signatures don't match
  - specifying override will have compiler warn us if signature doesn't match a superclass' virtual method
- why not declare everything as virtual for simplicity?
  ↳ e.g.
```
struct Vec {
  int x,y;
  void doSomething();
};
Vec v {1,2};
Vec2 w{3,4};
cout << sizeof(v); //8
cout <<sizeof(w); //16
```
```
struct Vec2 {
  int x,y;
  virtual void doSomething();
};
```
  - declaring doSomething as virtual doubles size of obj ∴ program consumes more RAM ∴ is slower in general

- extra 8B is storing **virtual ptr (vptr)**, which allows us to achieve dynamic dispatch w/ virtual fcns
  ↳ in MIPS, fcn calls use jalr ins, saves a register, & jumps PC to a specific fcn's mem address (which is hard-coded in that machine ins)
  ↳ w/ dynamic dispatch, which fcn to jump to could depend on user input & can't be hard-coded
  ↳ e.g.

```
struct Vec2 {
    int x, y;
    virtual void doSomething();
};
struct Vec3: public Vec2 {
    int z;
    void doSomething() override;
};
```




```
string choice;
cin << choice;
Vec2 * v;
if (choice == "vec2")
    v = new Vec2{...};
else
    v = new Vec3{...};
v → doSomething();
```

  - when we create a Vec2/Vec3, we know what type of obj we're creating so we can fill in appropriate vptr for that obj
  - now, in either case, we can simply follow vptr, get to vtable, & find fcn address for doSomething method

there's extra run-time cost in time it takes to follow vptr & across vtable
↳ c++ philosophy: don't pay for cost unless we ask for it

DESTRUCTORS

- e.g.

```
class X {
    int *a;
public:
    X(int n): a{new int[a]}{}
    ~X(){delete[] a;}
};
class Y: public X {
    int *b;
public:
    Y(int n, int m): X{n}, b{new int[m]}{}
    ~Y(){delete[] b;}
};
X x{5};
X* px = new X{5};
```

```
Y y {5, 10};
Y* py = new Y{5, 10};
X* pxy = new Y{5, 10};
```

↳ obj destruction seq:

1) dtor body runs
2) obj fields have their dtors run in reverse declaration order
3) superclass dtor runs
4) space is reclaimed

↳ since dtor is non-virtual, invoke ~X() not ~Y for pxy so arr b is leaked

- soln is to declare virtual ~X(); so delete pxy calls ~Y()

- unless we're sure a class will nvr be subclassed, then always declare dtors as virtual

↳ if we're sure, enforce it via final keyword

- e.g.

```
class X final {
  ...
};
```

→ program won't compile if anyone tries to subclass X

LECTURE 11

POLYMORPHIC BIG 5

- e.g.

```
class Shape {
public:
   virtual float getArea() const;
};
class Square: public Shape {
   float length;
public:
   Square(...) {...}
   float getArea() const override{
      return length * length;
   }
};
class Circle: public Shape {
   float radius;
public:
   Circle(..) {...}
   float getArea() const override {
      return PI * radius * radius;
   }
};
```

↳ if we don't provide implementation for Shape::getArea(), code won't link → error is "undefined ref to vtable error"

↳ could make Shape::getArea() return 0 or indicate "no area", but it's not natural
- avoid defn for Shape::getArea() entirely

↳ soln is to declare Shape::getArea() as **pure virtual fcn**, which is allowed to not have an implementation
-

```
class Shape {
public:
   virtual float getArea() const = 0;
};
```

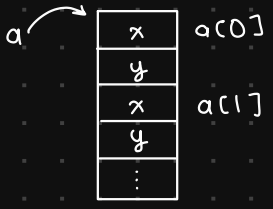
declares getArea() as pure virtual

- classes that declare a pure virtual fcn are called **abstract classes**
  ↳ abstract classes can't be instantiated as objs
- a class that overrides all of its parent's pure virtual fcns is called a **concrete class**
  ↳ concrete classes can be instantiated
- purpose of abstract class is to provide a **framework** to organize & define subclasses

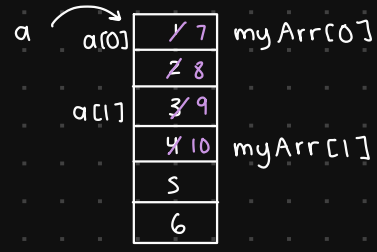
- e.g.

```
class Vec2 {
   int x, y;
public:
   Vec2(int x, int y): x{x}, y{y} {}
};
class Vec3: public Vec2 {
   int z;
public:
   Vec3(int x, int y, int z): Vec2{x, y}, z{z} {}
};
void f(Vec2* a) {
   a[0] = Vec2{7, 8};
```

a[1] = Vec2{9, 10};
}
Vec3 myArr[2] = {Vec3{1, 2, 3}, Vec3{4, 5, 6}};
f(myArr);
↳ f expects:



↳ what actually happens:



- data ends up misaligned
  - → all of Vec2{7, 8} is written into myArr[0]
  - → half of Vec2{9, 10} is written into myArr[0] & other half is in myArr[1]

↳ be careful when using arrs of objs polymorphically
↳ soln is to use arr of ptrs
- e.g. Vec3* myArr[2]

↳ other soln is to use vector of Vec3*s

- e.g.
  Text t{"Polymorphism", "Ross", 500, "c++"};
  Text t2{t};
  ↳ compiler still provides us a copy ctor that works as expected
- e.g. copy ctor:
  Text::Text(const Text& t): Book{t}, topic{t.topic} {}
  ↳ Book{t} calls copy ctor for Book portion of Text
  - t is const Text&, but it's implicitly converted to const Book&
- e.g. move ctor:
  Text::Text(Text&& t): Book{std::move(t)}, topic{std::move(t.topic)} {}
  ↳ t & t.topic are lvalues so we'd invoke copy ctor if we didn't use std::move
  ↳ t is rvalue ref so we know it's safe to steal title, author, length, & topic via std::move to invoke move ctors
- e.g. copy asgnt operator:

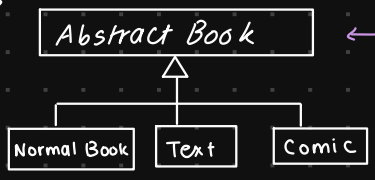
```
Text& Text::operator=(const Text& t) {
    Book::operator=(t);    // calls Book asgnt operator for Book portion of this
    topic = t.topic;
    return *this;
}
```

- e.g. move asgnt operator:

```
Text& Text::operator=(Text&& t) {
    Book::operator=(std::move(t));
    topic = std::move(t.topic);
    return *this;
}
```

- previous 4 implementations are what compiler gives by default

- e.g.
  ```
  Text t1{"Polymorphism", "Ross", 500, "c++"};
  Text t2{"Programming for Babies", "Laurier Prof", 100, "Python"};
  Book& br1 = t1;
  Book& br2 = t2;
  br2 = b1;
  cout << t2;
  ```
  ↳ title, author, & length are set, but topic remains unchanged
  ↳ Book::operator= is defined as non-virtual & we're calling operator= on a ref
    - use static type & call Book::operator= even though br1 & br2 are referencing Texts
  ↳ **partial asgnt problem**: some fields are copied, but not all
    - soln is to declare Book::operator= as virtual:
      ```
      class Book {
        ...
      public:
        ...
        virtual Book& operator=(const Book& other) {
          ...
        }
      };
      ```
- **usual signature** for Text copy asgnt operator:
  `Text& Text::operator=(const Text& rhs);`
  ↳ can't put override on end of this b/c signatures don't match in 2 places
    - return type: difference is ok
      → a subclass' overriden method can return subclass of virtual fcn's return type if it's ptr/ref
    - param type: must match exactly
      → signature must be:
      ```
      Text& Text::operator=(const Book& other) {
        ...
      }
      ```
      → can't access other's topic anymore b/c it's a Book & Books don't have topics
      → other is Book& so this is legal:
      ```
      // can ser Text to a Comic so Comic on RHS can be implicitly converted to const
      // Book&
      Comic c{...};
      Text t{...};
      t = c;
      ```
- **mixed asgnt problem**: setting subclass siblings to each other
- non-virtual operator= leads to partial asgnt
- virtual operator= leads to mixed asgnt
- one soln is to restructure Book hierarchy
  ↳



← in UML, abstract classes & virtual methods are italicized

LECTURE 12

POLYMORPHIC ASSIGNMENT PROBLEM

e.g.
```
class AbstractBook {
    string title, author;
    int length;
protected:
    AbstractBook& operator=(const AbstractBook& other) = default;
public:
    AbstractBook(...)?...{
    virtual ~AbstractBook() = 0;
```
↳ to make this abstract, need pure virtual method
- if no other methods make sense to be pure virtual, can always use dtor

e.g.
```
class Text: public AbstractBook{
    string topic;
public:
    Text(...) {...}
    //implicitly provided Big 5 so copy asgnt operator looks like. Text& operator=(const Text& o),
{
```
↳ mixed asgnt: operator= is non-virtual? implicitly provided copy asgnt operator only accepts Text

↳ partial asgnt:
```
Text t1{...};
Text t2{...};
AbstractBook& br1 = br;
AbstractBook& br2 = t2;
br2 = br1;   // doesn't compile b/c AbstractBook::operator= is protected
```
↳ only works since AbstractBook is abstract class
- if we set Book::operator= to protected, then we couldn't assign Books to one another

- in Text's dtor, following implicitly happens:
1) dtor body runs (empty)
2) obj fields are destructed in reverse declaration order
3) superclass dtor runs
4) space is reclaimed

↳ in 3, Text's dtor calls AbstractBook's dtor? we have problem b/c we've called method w/no implementation
- soln is to give it implementation: AbstractBook::~AbstractBook(){}
- AbstractBook is still abstract b/c only subclasses of classes of classes which define a pure virtual fcn may be concrete

- pure virtual methods don't have to be implemented, but they still can be
↳ require implementation if they're going to be called

recommendation: if we care abt asgnt, consider making superclass abstract

ERROR HANDLING

- STL vectors are dynamically allocated resizing arrs
↳ handles mem management so we don't screw it up
↳ can't stop everything
- e.g.
```
vector<int> v;
v.push_back(100);
cout << v[100] << endl;   // out of bounds access
```

- to handle errors:
  - ↳ **sentinel vals**: reserve some vals (e.g. -1, INT_MIN) to signal errors
    - reduces what we can return
    - not clear for general type T what vals we should pick as sentinels
  - ↳ **global vars**: create some global var that's set when error occurs
    - in C, there's int errno, which can be queried for errors w/ std fcns
    - limited #errors, might be overwritten
  - ↳ **bundle in struct**:

```
template <typename T> struct ReturnType {
    int errorcode;
    T* data;
};
```

    - best so far, but still not ideal
    - wrap return types in this struct so all return types are larger than needed
    - awkward to access data field

- all are approaches that C users use, but C++ has lang feature for dealing w/ errors: **exceptions**
  - ↳ e.g. v.at(100) fetches v[100] if val exists, otherwise there's exception
- e.g. **try-catch block**

```
try {
  cout << v.at(100) << endl;
}
catch (std::out_of_range r){
   cout << "Range error" << r.what() << endl;
}
```

  - ↳ r is obj of class type **std::out_of_range** (included in **<stdexcept>**)
  - ↳ .what() method returns str describing exception
  - ↳ forces programmer to deal w/ error b/c control flow jumps
  - ↳ **non-locality** of error handling: vector knows error happened but not how to fix it, we know how to fix it but not how error occurred
- to raise exception ourselves, use **throw** keyword
  - ↳ can throw any val, but <stdexcept> has objs for common scenarios like out_of_range, logic_error, invalid_argument
- when exception is raised, control flow stops
  - ↳ **stack unwinding**: search through stack upwards looking for handler for this type of exception
    - dtors are run for objs stored on stack during process of stack unwinding
    - if handler is found, jump to that point
    - if no handler is found, program crashes
    - e.g.

```
void f() {
   throw std::out_of_range{"f threw"};
}
void g() {f();}
void h() {g();}
int main() {
   try {h();}
   catch (std::out_of_range r){
      cout << r.what();
   }
}
```

      → main calls h, h calls g, g calls f, throws, stack unwinding thru g & h, then jump to catch block in main

- multiple errors may be handled via multiple catch blocks
  - ↳ e.g.

```
try {...}
catch (out_of_range r) {... }
catch (logic_error e) {...}
catch (invalid_argument i) {... }
catch(...) {...}
```

↑ catch-all syntax which catches any type of exception ? it's literally 3 dots

- one handler can also deal w/ part of error, then rethrow exception to allow some other fcn to deal w/ it
  - ↳ e.g.

```
void calculation(DataStructure& ds) {
    ...
    throw ds_error{...};
}
void DataStructureHandler(DataStructure& ds) {
    try {calculation(ds)}
    catch (ds_error e) {
        // fix DataStructure error
        throw prompt_input_error{...};
    }
}
int main() {
    DataStructure ds;
    string s;
    while (cin >> s) {
        try {
            DataStructureHandler(ds);
        } catch (prompt_input_error e) {
            cout << "Invalid input";
        }
    }
}
```

LECTURE 13

- e.g.
```
try {...}
catch (std::exception& e){
    ...
    throw;
}
```
  - ↳ use "throw" instead of "throw e" b/c "throw e" performs a copy to throw this exception
    - copy ctors (& any type of ctor) can't be virtual so static type is used for copy
  - ↳ if we throw a range_error & catch via std::exception& catch block:
    - "throw" rethrows range_error
    - "throw e" creates std::exception copied from range_error & we lose dynamic type

  generally, catch blocks should catch by ref to avoid copies
- never let dtor throw exception
  - ↳ default behaviour is that program immediately crashes
  - ↳ noexcept is tag associated w/methods (e.g. const) which states method will never raise exception
    - dtors are implicitly noexcept tagged
- can allow exns to be thrown from dtors by tagging them "noexcept(false)"
  - ↳ e.g.
```
class A {
public:
    ...
    ~A() noexcept(false) {... }
};
```
  - ↳ dangerous b/c if we throw exn, stack unwinding occurs & during this process, dtors are running for stack allocated objs
    - if one of dtors throws, now there's 2 active exns
      - → 2 active exns leads to program crash (can't be changed)

EXCEPTION SAFETY

- e.g.
```
void f(){
    MyClass m;
    MyClass* p = new MyClass{};
    q();
    delete p;
}
```
  - ↳ under normal circumstances, f does not leak mem
  - ↳ if q throws, we don't execute "delete p" & do leak mem
- exns significantly change control flow
  - ↳ no longer have guarantee of sequential execution
- e.g. fix f to handle mem leak
```
void f(){
    MyClass m;
    MyClass* p = new MyClass{};
    try {
        q();
    }catch(...) {
        delete p;
        throw;
    }
```

delete p;
}
↳ complaints abt fix:
- repeated code b/c "delete p" appears twice
- could get complicated w/ many pointers, fcn calls, etc.

- in other langs, some have "finally" clause which always run, either after successful try, successful catch, or before catch rethrows
  ↳ c++ doesn't have support for this; our only guarantee is that during stack unwinding, stack allocated objs have their dtors run
  - soln is to use stack allocated objs instead so there's no leak
    → can't work if we need ptrs (e.g. to achieve polymorphism)
- c++ idiom: RAII (resource acquisition is initialization)
  ↳ wrap ptr in a stack allocated obj that'll delete it for us during stack unwinding
  ↳ e.g.
```
{
  ifstream file{"file.txt"};
  ...
}
```
  - resource (file handler) is acquired in ctor (i.e. initialization)
  - resource is freed (i.e. closing file) in dtor
- apply RAII concept to dynamic mem
  ↳ std::unique_ptr<T> is in <memory> lib
  - contains T* passed via ctor
  - deletes T* in dtor
  ↳ e.g.
```
void f(){
  MyClass m;
  std::unique_ptr<MyClass> p{new MyClass{}};
  g();
}
```
  - if we leave f normally or during state unwinding, p's dtor runs & deletes heap mem either way
  - in btwn, can use p like regular pointer thanks to operator overloads
- generally, don't call ctor directly w/ pointer
  ↳ std::unique_ptr<T> p{new T{}};
  - new isn't paired w/ delete that we can see
  ↳
```
T* p = new T{};
std::unique_ptr<T> q{p};
std::unique_ptr<T> r{p};
```
  - causes double delete
  ↳ f(std::unique_ptr<T>{new T{}}, g());
  - one potential ordering in c++14 can result in mem leak (obscure):
    1) new T{}
    2) g()
    3) unique_ptr ctor
    4) f()

preferred alt is std::make_unique<T>(...)
↳ constructs T obj in heap w/ args ... & returns unique_ptr<T> pointing at this obj
- e.g.
```
unique_ptr<T> p = make_unique<T>(...);
unique_ptr<T> q = p;
```
↳ calls copy ctor to copy p into q
↳ doesn't compile b/c copying is disabled for unique_ptrs
  - can only be moved
  - achieved by setting copy ctor / asgnt operator to = delete

- unique_ptrs are good for rep ownership since when one obj dies, its unique_ptr field's dtor will run & clean up associated obj
  - ↳ for has-a rltnship, use raw ptrs / refs
    - access underlying raw ptr of smart ptr via .get()

LECTURE 14

SHARED POINTERS

- what if we want shared ownership, where some mem is deleted only if all ptrs pointing to it are deleted?
  - ↳ achieve via `std::shared_ptr<T>`
    - maintains a **ref count** which keeps track of how many shared ptrs point at this mem? when it hits 0, mem is freed
    - e.g.

```
class C {
  shared_ptr<C> p = make_shared<C>(...);   // args to create C obj
  if (...) {
    shared_ptr<C> q = p;      //copy ctor for shared ptrs
    ...
  } //dtor for q runs, ref count = 1
  ...
}; //dtor for p runs, ref count = 0, C obj is destroyed
```

- **problems** w/ shared ptrs:
  - ↳ susceptible if we use ctor as such:

```
C* cp = new C{...};
shared_ptr<C> p{cp};
shared_ptr<C> q{cp};   // double delete b/c q has no knowledge of p's existence
```

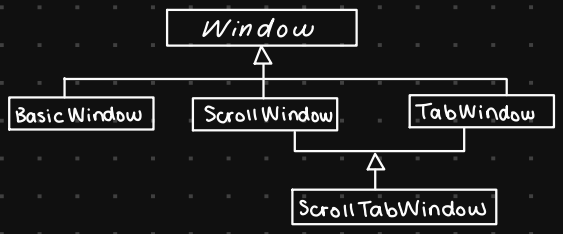
  - ↳ cyclical ref issue: if we have multiple shared ptrs that point at each other, we have issue that some ptrs may never be deleted
    - e.g.

```
class Node {
  vector<shared_ptr<Node>> neighbours;
};
```

- in general, shared ownership is rare
- when constructing programs, it suffices to use:
  - ↳ unique_ptr / obj field for **ownership / composition**
  - ↳ raw ptr / ref for **aggregation / has-a**
  - ↳ shared_ptr for **shared ownership**

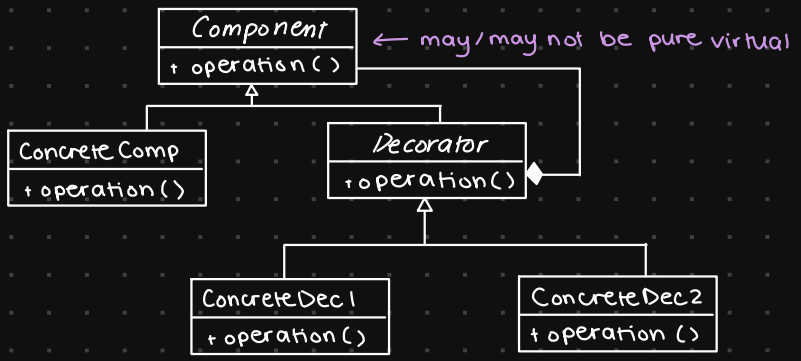
DESIGN PATTERNS

- **design patterns** provide some std good soln to common problem
  - a common problem is adding / removing behaviour at run-time w/ polymorphic class
  - ↳ e.g. windowing system GUI

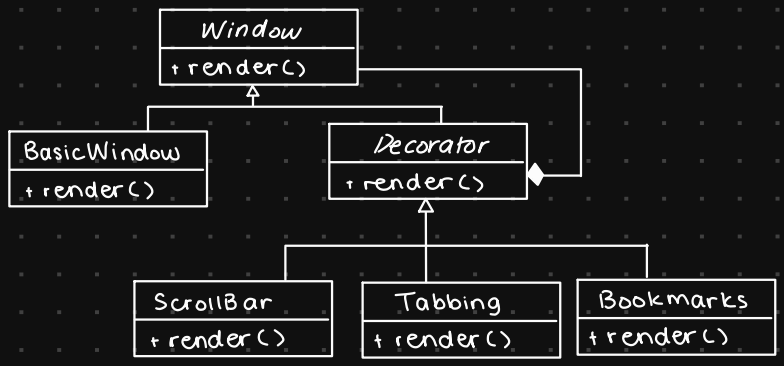


    - if we attempt to make a class for each possible feature for n features, there's 2^n configs
      → combinatorial explosion
  - ↳ soln is **decorator pattern**, which is **linked list of functionalities**
    - abstract component gives interface for our classes
    - concrete component implements "basic" version of interface
    - abstract decorator organizes decorators ↑ has-a concrete decorator / component

- concrete decorator implements interface, calls operation() on next obj in linked list
- diagram:

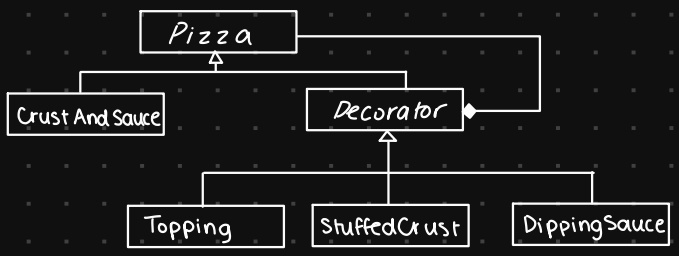


↳ e.g. windowing system GUI using decorator pattern




```
Window* w = new ScrollBar{new Tabbing{new BasicWindow{}}};
//ScrollBar → Tabbing → BasicWindow
w->render();
```

e.g. pizza



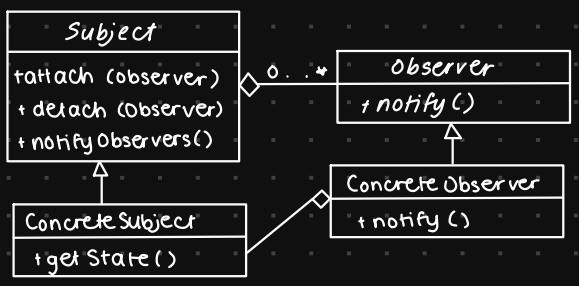

```
class Pizza {
public:
    virtual ~Pizza(){}
    virtual float price() const = 0;
    virtual string desc() const = 0;
};
class CrustAndSauce : public Pizza {
public:
    float price() const override {return 5.99;}
    string desc() const override {return "Pizza";}
};
class Decorator : public Pizza {
protected:
    Pizza *next;
public:
    Decorator(Pizza* next) : next{next} {}
   ~Decorator() {delete next;}
```

```
};
class Topping: public Decorator {
    string name;
public:
    Topping(const string& s, Pizza* p): Decorator{p}, name{s} {}
    float price() const override {
        return 0.75 + next->price();
    }
    string desc() const override {
        return next->desc() + " with " + name;
    }
};
class StuffedCrust: public Decorator {
public:
    StuffedCrust(Pizza *p): Decorator{p} {}
    float price() const override {
        return next->price() + 2.50;
    }
    string desc() const override {
        return next->desc() + " with stuffed crust";
    }
};
Pizza* p = new CrustAndSauce{};
p = new Topping{"cheese", p};
p = new Topping{"pepperoni", p};
p = new StuffedCrust{p};
cout << p->price << " " << p->desc << endl;
//output: 9.99 Pizza with cheese with pepperoni with stuffed crust
delete p;
```

LECTURE 15

OBSERVER PATTERN

- publisher / subscriber rltnship
  - ↳ **publisher**: generates some data, change in state
  - ↳ **subscribers**: can dynamically subscribe / unsubscribe from various publishers
    - should react when data is changed
- e.g. spreadsheet app
  - ↳ publishers: cells (subjects)
  - ↳ subscribers: charts (observers)
  - ↳ when cell changes, charts that are observing that cell must re-render & display new info
  - ↳ may have diff types of publishers / subscribers
    - e.g. diff charts require diff logic for re-rendering
- **diagram**:



  - ↳ **process** of what happens when state is changed:
    1) ConcreteSubject has its state updated
    2) notifyObservers is called either by ConcreteSubject or some controller (e.g. main fcn)
    3) notify is called on each Observer in Subject's Observer list
    4) this calls ConcreteObserver::notify, by assumption Observer::notify is pure virtual
    5) ConcreteObserver calls getState on ConcreteSubject & uses info as necessary
- e.g. Twitter clone
  - ↳ Tweeters are subjects
  - ↳ Followers are observers & can only follow 1 Tweeter
  - ↳ class Subject {

```
    vector <Observer*> observers;
  public:
    void attach (Observer* o) { observers.push_back(o); }
    void detach (Observer* o) { // remove from observers }
    Void notifyObservers() {
        for (Observer* o : observers)
            o -> notify();
    }
    virtual ~Subject() = 0;   // make this pure virtual so Subject is abstract
};
Subject::~Subject() {}
class Observer {
public:
    virtual void notify() = 0;
    virtual ~Observer() {}
};
class Tweeter: public Subject {
    ifstream in;
    string tweet;
public:
    Tweeter (const string& fileName): in{fileName} {}
```

```
    bool tweet() {
        getline(in, tweet);
        return in.good();
    }
    string getState() {return tweet;}
};
class Follower: public Observer {
    Tweeter* iFollow;
    string myName;
public:
    Follower(Tweeter* t, string s): iFollow{t}, myName{s} {
        iFollow->attach(this);
    }
    ~Follower() {
        iFollow->detach(this);
    }
    void notify() {
        string tweet = iFollow->getState();
        if (tweet.find(myName) != string::npos) {
            cout << "They said my name!" << endl;
        }
        else {
            cout << ":(" << endl;
        }
    }
};
int main() {
    Tweeter elon {"elon.txt"};
    Follower joe {&elon, "joe"};
    Follower jane {&elon, "jane"};
    while (elon.tweet()) {
        elon.notifyObservers();
    }
}
```

- only works b/c destruction happens in reverse order
- design patterns we demo are w/ simplistic implementations
  - → only get val at scale
  - → can be multiple types of subjects/observers
  - → other variants also exist (e.g. passing Subject& via notify method)

CASTING

- **casting** allows us to take 1 type & treat it as another
- e.g. in C:

```
Node n;
int* ip = (int*)&n;
```

(casting operator, which means to treat as int*)

↳ dangerous & generally avoid casting unless necessary b/c it subverts type system

- use 1 of **4 C# casts** instead of C-style casting

1) **static_cast**: well-defined conversions btwn 2 types
   - e.g.

```
void q(float f);
void q(int n);
float f;
q(static_cast<int>(f));   // calls int ver of q
```

   - e.g.

```
Book *b = ...;
Text* t = static_cast<Text*>(b);
```
→ undefined behaviour if b is not acc pointing at a Text

2) `reinterpret_cast`: allows for poorly-defined, implementation dependent cast
- most uses are undefined behaviour
- e.g.
```
Rational r;
Node* n = reinterpret_cast<Node*>(&r);
```

3) `const_cast`: only type of cast that can remove constness
- e.g.
```
void q(int* p);
```
→ know q doesn't modify int pointed to by p in any way
→ if we have const int* we'd like to call q with. compiler will prevent us from calling q b/c it might modify p
→ soln:
```
void f(const int* p){
    q(const_cast<int*>(p));
}
```
- should be avoided in general
- e.g. add consts to legacy codebase that doesn't use const anywhere ∴ make program const-correct
→ issue is `const-poisoning` b/c adding const in 1 location often means we must add it to other locations to allow it to continue to compile
- use to bridge btwn const-correct ∴ non-const-correct parts of program
→ make small independent parts of program const-correct ∴ use const_cast to allow program to compile as work is done

LECTURE 16

CASTING

- 4th type of casting is `dynamic_cast`: used for safely casting btwn pointers/refs in inheritance hiearchy
  - ↳ e.g.

```
Book *pb = ...;
static_cast<Text*>(pb)->getTopic();
```

    - only safe if pb acc points to Text
    - instead:

```
Book *pb = ...;
Text *pt = dynamic_cast<Text*>(pb);
if (pt) cout << pt->getTopic();
else cout << "Not a Text!";
```

      → if cast succeeds, then pt points at Text
      → else, pt is nullptr

  - ↳ can also be used on refs
    - e.g.

```
Text t{...};
Book& br = t;
Text& tr = dynamic_cast<Text&>(br);
```

    - if dynamic_cast fails w/ ref, throw `std::bad_cast` exception
- there's smart ptr versions of each cast:
  - ↳ static_pointer_cast
  - ↳ const_pointer_cast
  - ↳ dynamic_pointer_cast
- casting shared_ptrs to other shared_ptrs allows us to make decisions based on `RTTI (run-time type info)`
  - ↳ e.g.

```
void whatIsIt(shared_ptr<Book> b) {
    if (dynamic_pointer_cast<Text>(b))
        cout << "Text";
    else if (dynamic_pointer_cast<Comic>(b))
        cout << "Comic";
    else
        cout << "Normal Book";
}
```

    - poor OOD so don't do this
- note dynamic_cast only works if there's at least 1 virtual method
- e.g. polymorphic asgnt problem where we considered making operator= virtual

```
Text t1, t2;
Book& b1 = t1;
Book& b2 = t2;
b1 = b2;
```

if operator= is non-virtual, partial asgnt
if operator= is virtual, mixed asgnt

  - ↳ make operator= virtual in Book:

```
Text& Text::operator=(const Book& other) {
    Text& tother = dynamic_cast<const Text&>(other);   // throws if other is not Text
    if (this == &other)
        return *this;
    Book::operator=(other);
    topic = tother.topic;
    return *this;
}
```

MEASURES OF DESIGN QUALITY

- how to evaluate quality of code when we're not just following particular design pattern?
  - ↳ code smells: warning signs that suggest potential problem
    - refactoring is needed
  - ↳ coupling & cohesion: how classes interact
  - ↳ SOLID design principles

- coupling is description of how strongly diff modules depend on one another
  - ↳ low is where:
    - fcn calls w/primitive args/results
    - fcn calls w/structs & arrs as args/results
    - modules affect each other's control flow
    - modules sharing global data
  - ↳ high is where modules have access to each other's implementation (i.e. friends)
- we desire low coupling b/c it's easier to reason abt & make changes to program
  - ↳ if we put everything in 1 class, it's super low coupling
    - counterbalancing force is cohesion

cohesion is how closely we want parts of a module to relate to one another
- ↳ low is where:
  - parts of module are completely unrelated
    - → e.g. <utility>
  - module has unifying common theme
    - → e.g. <algorithm>
  - elmts manipulate state over lifetime of obj
    - → e.g. <fstream>
- ↳ high is where elmts are cooperating to perform exactly 1 task
  - e.g. put every fcn in its own class
- strive for low coupling & high cohesion
- e.g. what Is It fcn exhibits high coupling w/Book hierarchy
  - ↳ any new classes in Book hierarchy necessitate changes to this fcn

SOLID DESIGN PRINCIPLES

- acronym of 5 principles, whose purpose is to reduce software rot (i.e. property that long lasting codebases become more difficult to maintain)
  - ↳ **S**ingle Responsibility Principle
    Open/Closed Principle
    Liskov Substitution Principle
    Interface Segregation Principle
    Dependency Inversion Principle

- Single Responsibility Principle (SRP) means a class should only do 1 thing, not several
- change to program specs require change to source code
  - ↳ if changes to 2+ diff parts of spec require changes in same class, then SRP is violated
- e.g.

```
class ChessBoard {
    ...
    ...   cout << "Your Move";
};
```

- ↳ ChessBoard should not print to screen b/c what if we wanted to instead print to file
  - can't use same class w/o replacing couts w/ printing to file
- ↳ better design:

```
class ChessBoard {
    ostream& out;
```

```
    istream& in;
  public:
    ChessBoard(ostream& out, istream& in): out{out}, in{in}{
      ...
      ... out << "Your Move";
      ...
  };
```

- more flexible, but still has issues b/c what if we want:
  - → graphical design
  - → change lang
  - → communicate over Internet
- low cohesion, which violates SRP
  - → ChessBoard is doing multiple things like manipulating state, implementing logic, & controlling rendering to screen

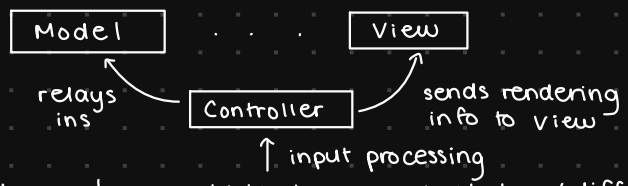
↳ best design is for Chessboard to communicate via results, params, & exceptions

- allow other classes to handle communication w/ user

LECTURE 17

SINGLE RESPONSIBILITY PRINCIPLE

- should main be performing communication?
  - ↳ no b/c it limits reusability
- use Model-View-Controller (MVC) architecture
  - ↳ model handles logic & data of program
  - ↳ view handles program output/display
  - ↳ controller handles input & facilitates control flow btwn classes
  - ↳

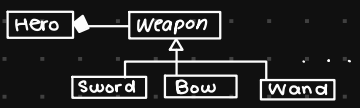


- model may have multiple views or just 1 w/ diff types (e.g. text, graphical view)
  - ↳ doesn't need to know anything abt state/implementation of views
  - ↳ sometimes structure interactions btwn models & views via Observer pattern
- controller mediates control flow
  - ↳ may encapsulate things like turn-taking / some portion of game rules
    - some of this logic may go in model instead
  - ↳ may communicate w/ user for input
    - sometimes done by view
- by decoupling presentation, input, & data/logic, we follow SRP & promote reuse in programs
  - ↳ however, some parts of specs may be unlikely to change so adding layers of abstraction just to follow SRP may not be worthwhile

OPEN/CLOSED PRINCIPLE

- classes should be open to extension, but closed to modification
  - ↳ changes to program should occur primarily by writing new code, not changing existing code
- e.g.

  - ↳ what if we wanted Hero to use diff type of weapon
    - requires us to change all places we ref Sword class
      - → lots of modifications & not much extension
    - fix:



- closely related to concept of inheritance & virtual fcns
  - ↳ e.g.

```
int CountHeavy(const vector<Book*>& v);
    int count = 0;
    for (auto p: v) {
        if (p->isHeavy())
            ++count;
    }
    return count;
}
```

    - open to extension b/c we can add more fcnality by defining new types of Books
    - closed to modification b/c countHeavy never needs to change now that it's been written
- e.g. whatIsIt is not open to extension b/c can't get new behaviour w/o modifying source code
  - ↳ not closed to modification either

open / closed principle is an ideal
↳ writing a program will require modifications at some point
↳ plan for things that are most likely to change & account for those

LISKOV SUBSTITUTION PRINCIPLE

↳ enforces that public inheritance should model **is-a rltnship strictly**
- if class B inherits from class A, can use ptrs/refs to B objs in place of ptrs/refs to A objs
  ↳ c++ gives us this
  ↳ Liskov sub is stronger b/c not only can we perform sub, but we should be able to do so at any place in the program w/o affecting its correctness
- if invariant is true for class A, then it should also be true for class B
- if invariant is true for method A::f & B overrides f, then it must be true for B::f
  ↳ if B::f **overrides** A::f, if A::f has precond P & postcond Q, then B::f should have precond P' st P ⇒ P' & postcond Q' st Q'⇒Q
    - i.e. B::f needs weaker precond & stronger postcond
- **contravariance** problem happens whenever we have binary operator where other param is same type as *this
  ↳ e.g.

```
class Shape{
public:
  virtual bool operator==(const Shape& other) const;
};
class Circle{
public:
  bool operator==(const Shape& other) const override;
};
```

  - must take in same param when overriding
  - c++ enforces this, which acc enforces LSP:
    1) a Circle is-a Shape
    2) a Shape can be compared w/ other Shapes
    3) a Circle can be compared w/ any other Shape
  - to satisfy LSP, must support comparison btwn diff types of Shapes
    → e.g.

```
bool Circle::operator==(const Shape& other) const{
  if (typeid(other) != typeid(Circle))
    return false;
  const Circle& cother = static_cast<const Circle>(other);
  ...
}
```

- **typeid** returns std::type into objs that can be compared
  ↳ e.g. diff btwn typeid & dynamic_cast
    - dynamic_cast: is other a Circle or subclass of Circle
    - typeid: is other exactly a Circle
  ↳ typeid uses dynamic_type so long as the class has at least 1 virtual method
- e.g. is square a rectangle?

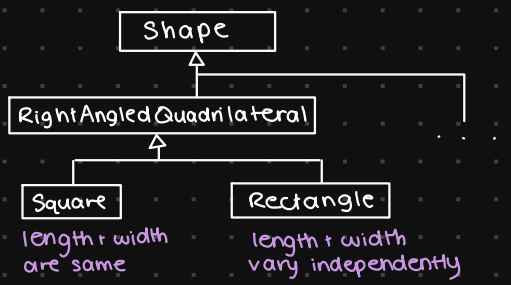
```
class Rectangle{
  int length, width;
public:
  Rectangle(int l, int w): length{l}, width{w}{}
  int getLength() const;
  int getWidth() const;
  virtual void setWidth(int w) {width = w;}
  virtual void setLength(int l) {length = l;}
  int area() const { return length * width;}
```

```
};
class Square : public Rectangle {
public:
    Square(int s): Rectangle{s, s} {}
    void setWidth(int w) override {
        Rectangle::setWidth(w);
        Rectangle::setLength(w);
    }
    //same for setLength
};
int f(Rectangle& r) {
    r.setLength(10);
    r.setWidth(20);
    return r.area();   //should return 200
}
Square s{100};
f(s);  //gives us 400
```

↳ issue is that we expect our postcond for Rectangle::setWidth to be that width is set ? nothing else changes, which is violated by Square class

↳ violates LSP ? does not satisfy is-a rltnship so a square is not a rectangle

↳ soln: restructure inheritance hiearchy to make sure LSP is respected



- to prevent LSP violations, restrain what our subclasses can do ? only allow them to customize what is truly necessary

↳ use design pattern, template method pattern

- provide structure / recipe / template for what method will do ? allow subclasses to customize portions of this method by performing overrides
- generally, subclass should be able to modify some parts of method's behaviour, but not all

LECTURE 18

TEMPLATE METHOD PATTERN

- e.g. in vid game, we have 2 types of Turtles : Red Turtle ? Green Turtle

```
class Turtle{
   virtual void drawShell() = 0;
   void drawHead() {☺;}
   void drawLegs() {0000;}
public:
   void draw {
      drawHead();
      drawShell();
      drawLegs();
   }
};
class RedTurtle: public Turtle{
   void drawShell() override {
       ;
   }
};
class GreenTurtle: public Turtle{
   void drawShell() override {
       ;
   }
};
```

  - ↳ draw method is public ? non-virtual
    - nobody who subclasses Turtle can override draw
    - GreenTurtle ? RedTurtle can define draw, but it won't be override so it won't be called if we use Turtle polymorphically
  - ↳ drawShell is private, but still can be overridden
    - access specifiers only describe when methods can be called, not how they may be overridden

NON-VIRTUAL IDIOM

- public virtual methods try to do 2 things at once:
  - ↳ public : provides interface to client
    - will uphold invariants
    - respect pre/post-conditions
    - expected to do its job
  - ↳ virtual : provides interface to subclasses
    - virtual methods may be customized while obj is being used polymorphically
    - provides "hooks" to allow customization of behaviour
- satisfying responsibilities of public ? virtual simultaneously is difficult
  - ↳ how to ensure public virtual method will satisfy its invariants, pre/post-conds, or even do its job?
  - ↳ what if we want to add more code to public virtual method w/o changing interface?
- non-virtual idiom (NVI) states:
  - ↳ all public methods are non-virtual
  - ↳ all virtual methods should be private/protected
  - ↳ exception is dtor
  - ↳ e.g.

```
// no NVI
class DigitalMedia{
```

```
    public:
      virtual ~DigitalMedia(){}
      virtual void play()=0;
  };
  //w/ NVI
  class DigitalMedia {
    virtual void doPlay()=0;
  public:
    virtual ~DigitalMedia(){}
    void play(){
      doPlay();
    }
  };
```

- can add code that must run for all types of DigitalMedia before/after doPlay call
  - → could add copyright checking before doPlay
  - → update a play count after doPlay so subclasses don't have to worry abt maintaining this invariant

allows for more flexibility b/c we can provide more "hooks" for customization simply by adding more private virtual methods

↳ e.g. showArt() before doPlay() is also private virtual
- could display poster for movie, album cover for song, etc.

↳ don't need to change public interface which is good for minimal recompilation ? open/closed principle

- in general, it's easier to constrain what subclasses do from beginning, as opposed to wrestling back control
  ↳ supports Liskov Sub Principle
- any decent compiler will optimize extra fcn calls so no cost at run-time

INTERFACE SEGREGATION PRINCIPLE

- no code should depend on methods it doesn't use
- prefer many small interfaces over 1 larger, more complicated interface
- if class has many fcnalities, each client of class should only see fcnality it needs
- e.g

```
//no NVI
class Enemy{
 public:
   virtual void strike();     //used in combat process
   virtual void draw();       //used by UI
};
class UI{
  vector<Enemy*> enemies;     //all enemies UI might need to draw
  ...
};
class Battlefield{
  vector<Enemy*> enemies;     // all enemies that might perform combat
  ...
};
```

↳ imagine we make change to drawing interface
- battlefield.cc must still recompile, even though it's not using any part of drawing interface

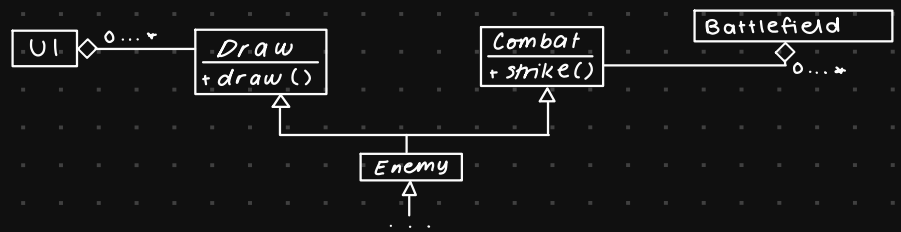
↳ needless coupling btwn UI ? Battlefield via Enemy

↳ 1 soln is multiple inheritance:

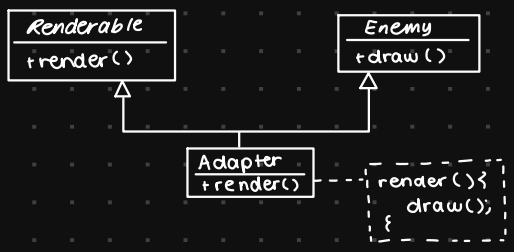
```
class Enemy: public Draw, public Combat{};
class Draw{
public:
```

```
    virtual void draw() = 0;
};
class Combat {
public:
    virtual void strike() = 0;
};
class UI {
    vector <Draw*> enemies;
};
class Battlefield {
    vector <Combat*> enemies;
};
```

- UML looks like:



- soln is somewhat similar to **adapter pattern**, which is used when a class provides an interface diff from one we need
  - ↳ e.g. library for our window expects objs to inherit from Renderable & provide render method
    - don't want to change all of inheritance hierarchy or all draw calls to render b/c it violates open/closed & it's a pain
    - use Adapter class:



→ satisfies Renderable interface by calling methods we've alr defined

→ might not even need Adapter to provide draw method anymore, just render (could use private inheritance)

- e.g.

```
class A:
  int x:
protected:
  int y;
public:
  int z:
};
```

↳ under **protected inheritance**: class B: protected A{...}

- x remains inaccessible
- y remains protected
- z becomes protected so it can only be accessed in B & subclasses of B

↳ under **private inheritance**: class B: private A{...} or class B: A {...}

- x remains inaccessible
- y & z become private so they can only be accessed in B methods

- protected & private inheritance are not is-a (i.e. specialization) rltnships

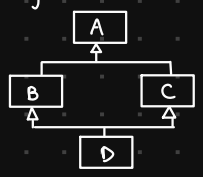
LECTURE 19

MULTIPLE INHERITANCE

- can't use classes w/ private / protected inheritance polymorphically
  - ↳ e.g. `A* p = new B{...};` //only legal under public inheritance
- multiple inheritance can have same tricky semantics
  - ↳ e.g.

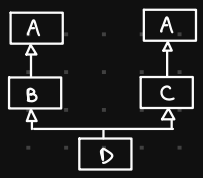
```
class A1 {
public:
  void foo() { cout << "A1::foo" << endl; }
};
class A2 {
public:
  void foo() { cout << "A2::foo" << endl; }
};
class B: public A1, public A2 {};
B b{};
b.foo();  // what should happen?
```

  - doesn't compile
  - we must **disambiguate** using `b.A1::foo()` or `b.A2::foo()`
- another issue of multiple inheritance is **diamond problem** (i.e. deadly diamond of death)
  - ↳ shared base class
  - ↳ e.g.




```
struct A { int a = 1; };
struct B : A {};
struct C : A {};
struct D : B, C {};
D dOB;
dOB.a = 2;  // doesn't compile either
```

} don't need to specify public b/c structs have public inheritance by default

  - D is-a B & is-a C so it has 2 a fields
  - 

  - must disambiguate: which field we're talking abt, one from B portion or one from C portion?
    - → `dOB.B::a = 1;`
      `dOB.C::a = 2;`
  - to disable this behaviour & have only 1 copy of A in hierarchy, use **virtual inheritance**
    - → e.g.

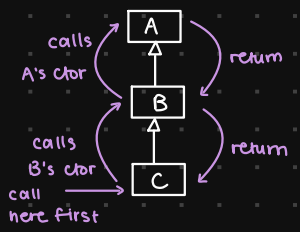
```
struct B: virtual A {...};
class C: public virtual A {...};
```

    - → keyword "virtual" indicates base class will be shared among all others that inherit from it virtually in inheritance hierarchy
    - → `dOB.a` becomes unambiguous b/c a field is shared btwn both portions of obj

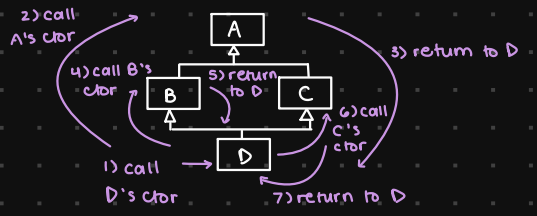
- e.g.

```
class A {
public:
```

A(int x){...}
};
class B: public virtual A{
public:
B(...): A{...}{...}
};
class C: public virtual A{
public:
C(...): A{2}{}...{
};
class D: public B, public C{...};
↳ ctors for virtual bases must run before any other ctors
↳ typically:



↳ virtual inheritance:

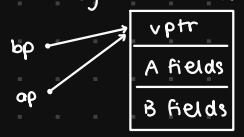


↳ fixed:

```
class A{
public:
    A(int x): ...}...{
};
class B: public virtual A{
public:
    B(): ... {...}...{
};
class C: public virtual A{
public:
    C(): ... {...}...{
};
class D: public B, public C{
public:
    D(): A{5}, B{}, C{}...{
};
```

(call A's ctor; could be left out)

- e.g. obj layout w/ multiple virtual inheritance
// not multiple inheritance
class A {...}{};
class B: public A{...}{};
B* bp = new B{...}{};
A* ap = bp;
↳ B obj looks like:



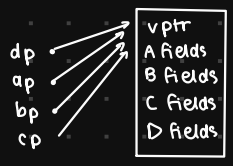
↳ to treat B* as A*, create ptr copy & ignore B fields

↳ same strategy can't be used for multiple virtual inheritance

- e.g.

```
D* dp = new D?...;
A* ap = dp;
B* bp = dp;
C* cp = dp;  // doesn't look like C obj if we point at same location
```

→



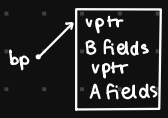
- in g++ / clang, a D obj looks like:

↳ virtual base is stored at end of obj

↳ e.g.

```
B* bp = ...,
cout << bp->a << endl;
```

- if bp pts at D obj, see above mem diagram
- if bp pts at B obj:



- if ptr'ing at D obj, bp->a is 40B below ptr address
- if ptr'ing at B obj, bp->a is 24B below ptr address

↳ finding superclass fields depends on dynamic type

- soln is to store offset to superclass fields in vtable

↳ note that D obj doesn't look like A obj, C obj, & D obj simultaneously, but portions of it does

↳ ptr asgnt can change where ptr is pting to

- e.g.

```
D* dp = new D?...;
A* ap = dp;
```

↑ pts at just A portion

← pts at top of D obj

- static_cast / dynamic_cast will adjust ptr locations as necessary

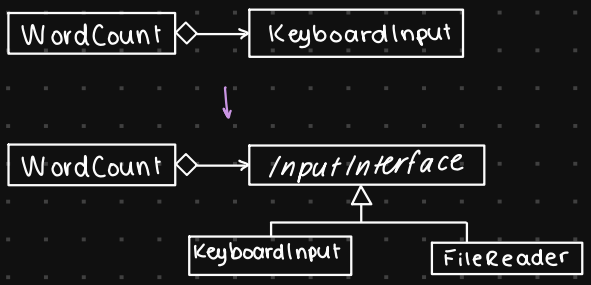
↳ reinterpret_cast won't so it's dangerous

- both Google c++ style guide & ISOCPP recommend when using multiple inheritance, interface classes are best
- interface classes define all methods as pure virtual & contain no states (i.e. fields)

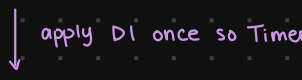
LECTURE 20

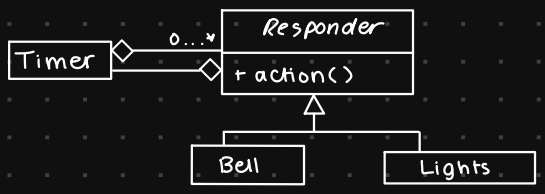
DEPENDENCY INVERSION PRINCIPLE

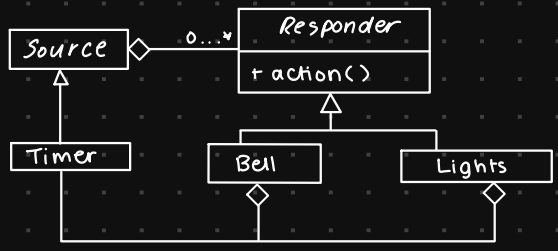
- D in SOLID principles
- dependency inversion principle means high-lvl modules shouldn't depend on low-lvl modules
  - ↳ each should depend on abstraction
    - abstract classes shouldn't depend on concrete classes
  - ↳ e.g. Microsoft Word word counter relies on some sort of keyboard input
    - breaks DI principle b/c counting words is high-lvl & getting input is low-lvl



  - ↳ e.g. timer w/ lights / bells that go off







    - end UML is Observer pattern
      - → Sources are subjects
      - → Responders are observers
    - could apply DI again to intro another layer of abstraction btwn Timer & Bell / Lights, but too many layers of abstraction creates needless complexity

VISITOR PATTERN

- how to write method that depends on 2 polymorphic types?
- e.g.

  - ↳ might want diff behaviours for each combo of concrete classes
  - ↳ class Weapon {

```
public:
```

```
    virtual void strike(Enemy& e) = 0;
};
```

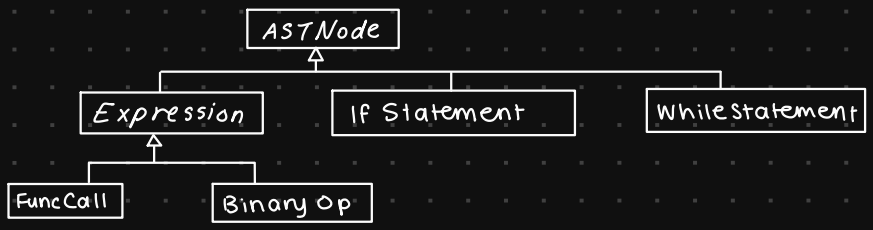
- can override in Rock & Stick but we don't know what Enemy type we're hitting
  → same problem exists in reverse if we implement inside of Enemy
- soln is **visitor pattern**, where we combine overloading w/ overriding:

```
class Enemy {
public:
    virtual void beStruckBy(Weapon& w) = 0;
    ...
};
class Monster : public Enemy {
public:
    void beStruckBy(Weapon& w) {
        w.strike(*this);
    }
};
class Turtle : public Enemy {
public:
    void beStruckBy(Weapon& w) {
        w.strike(*this);
    }
};
class Weapon {
public:
    virtual void strike(Monster& m) = 0;
    virtual void strike(Turtle& t) = 0;
};
class Rock : public Weapon {
public:
    void strike(Monster& m) override {
        cout << "Rock + Monster" << endl;
    }
    void strike(Turtle& t) override {
        cout << "Rock + Turtle" << endl;
    }
};
class Stick : public Weapon {
public:
    void strike(Monster& m) override {
        cout << "Stick + Monster" << endl;
    }
    void strike(Turtle& t) override {
        cout << "Stick + Turtle" << endl;
    }
};
Enemy* e = ...;
Weapon* w = ...;
e->beStruckBy(*w);
```

1) beStruckBy is virtual fcn so we're either calling Monster::beStruckBy or Turtle::beStruckBy (runtime)
2) call w strike on Weapon& → since strike is virtual fcn, we call Rock::strike or Stick::strike (runtime)
3) to determine if we're calling Rock::strike(Turtle&) or Rock::strike(Monster&), we know type of **this** (compile time)

→ if this is Turtle*, call Turtle version
→ if this is Monster*, call Monster version

- another use of visitor pattern is to add behaviour / functionality to classes w/o changing classes themselves
  ↳ e.g. compiler design



- might want to collect info abt AST
- could add virtual method in ASTNode / override in concrete classes
  ↳ lots of changes scattered over many classes may be difficult to understand / harder to implement
- instead provides ASTNodes w/ accept method / take in polymorphic ASTVisitor:

```
class ASTNode {   // like Enemy
public:
    virtual void accept(ASTVisitor& a) = 0;   // like beStruckBy
};
class IfStatement: public ASTNode {
public:
    void accept(ASTVisitor& a) {
        a.visit(*this);
        // call accept on subnodes in IfStatement (bool expr, block of stmts in IfStatement)
    }
};
class ASTVisitor {   // Weapon
public:
    virtual void visit(IfStatement& i) = 0;
    virtual void visit(WhileStatement& w) = 0;
    ...
    // for each ASTNode type
};
```

→ write diff types of visitors to work on parse tree
- StringVisitor: collect str literals in program that must go into data segment
- ScopeVisitor: match vars w/ associated scope
- CodeGenerationVisitor: generate assembly for each given node

→ soln improves cohesiveness b/c all code for particular visitor is located in 1 class
- new visitors are easy to create / can be made w/ 0 changes to ASTNode class

LECTURE 21

CRTP

too much code in visitor pattern
- ↳ there's going to be a lot of methods b/c if we have n subclasses of A & m subclasses of B, there's nm methods
- ↳ boilerplate code has to be in every Derived Enemy:

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

- must be in every single DerivedEnemy
- can't put in Enemy:

```
class Enemy{
public:
    void beStruckBy(Weapon& w){
        w.strike(*this);   //just an Enemy
    }
};
```

→ doesn't work b/c type of this is wrong & it doesn't tell us what dynamic type is

- soln to fix boilerplate code is CRTP (curiously recurring template pattern)
  - ↳ template superclass w/ type param
    - inherit & sub in direct class type
  - ↳ `template<typename T> class Base{ ... };`

atp, we know Derived is class so we can use T in Base as if we had provided fwd declaration

```
class Derived: public Base<Derived>{
    ...
};
```

- e.g. fixing boilerplate code

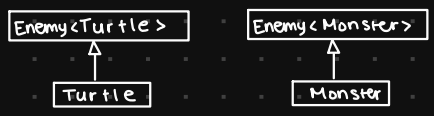
```
template<typename T> class Enemy{
public:
    void beStruckBy(Weapon& w){
        w.strike(*static_cast<T*>(this));
    }
};
class Monster: public Enemy<Monster> {...};
class Turtle: public Enemy<Turtle> {...};
```

- ↳ this sort of works:

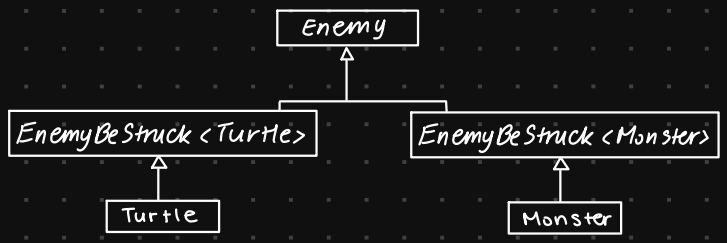
```
Weapon* w = ...;
Turtle t{...};
t.beStruckBy(*w);  // calls Enemy<Turtle>::beStruckBy
```

- casts this from Enemy<Turtle>* to Turtle*, which allows us to override into Rock/Stick::Strike(Turtle&)

- issue is that we have diff superclasses for each Enemy
  - ↳



- ↳ since Enemy<Turtle> & Enemy<Monster> are diff classes, can no longer use Enemys polymorphically
  - e.g. no vector<Enemy*> allowed
- soln is to add another layer of inheritance



↳ e.g.

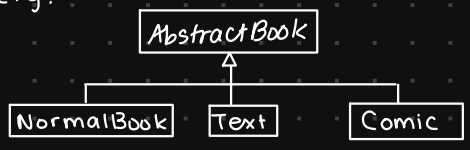
```
class Enemy {
public:
    virtual void beStruckBy(Weapon& w) = 0;
    virtual ~Enemy() {};
};
template<typename T> public EnemyBeStruck: public Enemy {
public:
    void beStruckBy(Weapon& w) override {
        w.strike(*static_cast<T*>(this));
    }
    virtual ~EnemyBeStruck() = 0;
};
template<typename T> EnemyBeStruck<T>::~EnemyBeStruck<T>() {}
class Turtle: public EnemyBeStruck<Turtle> {...};
class Monster: public EnemyBeStruck<Monster> {...};
```

- now, we have public interface by which all of our concrete Enemys follow ; they can call all beStruckBy weapons
- use virtual method in Enemy to resolve beStruckBy to either EnemyBeStruck<Turtle> / <Monster>
- then, just use static_cast to T*
  → e.g.

```
Weapon* w = ...;
Enemy* e = new Turtle{...} / new Monster{...};
e->beStruckBy(*w);
```

- another problem CRTP can solve is **polymorphic cloning**

↳ e.g.




```
AbstractBook* b = ...;
//want deep copy of whatever b ptrs to
AbstractBook* b2 = new AbstractBook{*b};   // can't do this
```

- attempts to create AbstractBook by invoking its ctor
- wrong b/c:
  1) AbstractBook is abstract ; can't instantiate these objs
  2) we ignore what we're actually pting at even though we actually want to invoke ctor that depends on dynamic type of b

↳ provide **virtual clone** method to solve this:

```
class AbstractBook {
public:
    virtual AbstractBook* clone() = 0;
    virtual ~AbstractBook() {}
};
// Comic ; NormalBook are similar
AbstractBook* b = ...;
AbstractBook* b2 = b->clone();
```

```
// use copy ctor in each of our clone methods to simplify implementation
class Text: public AbstractBook {
public:
  Text* clone() override {
    return new Text{*this};
  }
};
```

- exact same code in NormalBook & Comic, just type of this's ctor which are changing so we use CRTP

- e.g. using CRTP

```
class AbstractBook {
public:
  virtual AbstractBook* clone() = 0;
  virtual ~AbstractBook() {}
};
template<typename T> class BookCloneable: public AbstractBook {
public:
  T* clone() override {
    return new T{*static_cast<T*>(this)};
  }
  ~BookCloneable() = 0;
};
template<T> BookCloneable<T>::~BookCloneable<T>() {}
class Text: public BookCloneable<Text> {...};
class Comic: public BookCloneable<Comic> {...};
AbstractBook* b = new Text{...} / new Comic{...};
AbstractBook* b2 = b->clone();
```

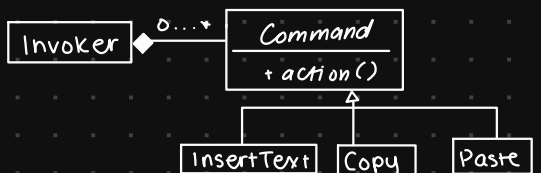
↳ b->clone() is virtual so if b pts at Comic, we call BookCloneable<Comic>::clone
- static-cast this into Comic*
- invoke Comic copy ctor w/ Comic&

↳ provided for all subclasses & reduces boilerplate code

COMMAND PATTERN

- abt using objs to encapsulate behaviour of some action with a system
  - ↳ e.g. IDE using MVC
    - might have Model/Controller rltnship where Controller calls Model methods
      - → Model::insertText
      - → Model::copy
      - → Model::paste
      - → Model::changeSyntaxHighlighting
    - decent design but doesn't allow some features like:
      - → macros (e.g. replaying seq of ins)
      - → undo/redo
- command pattern: instead of manipulating Model directly, pass cmds to Invoker

↳



↳ Command obj has whatever info it needs to perform its given action (maybe Model, maybe subobjs within Model)
1) Controller creates Command objs & supply them w/ info needed
2) sends abstract Commands to Invoker for processing
3) Invoker calls each action method to execute cmd

↳ Invoker can maintain additional state stack of cmds for undo/redo & mapping of macro keyword to

list of cmds

LECTURE 22

EXCEPTION SAFETY

- reasoning abt programs w/ exceptions is tricky
  - ↳ e.g.

```
void f(){
  int* p = new int{247};
  q();
  delete p;   // mem is leaked if q throws
}
```

- can be more specific abt how safe our program is given an exception is thrown
- 4 lvls of exception safety:
  1) **no exception safety**: if expn is thrown, no guarantees abt program state
     - obj may have invalid mem, mem is leaked, & program crashes
  2) **basic guarantee**: if expn is thrown, program is in valid, but unspecified, state
     - e.g. class invariants are maintained, no mem is leaked
  3) **strong guarantee**: if expn is thrown, program reverts back to its state before method was called
     - e.g. for vector: emplace_back, either it succeeds or if expn is thrown, vector is in its prior state
  4) **nothrow guarantee**: expns are nvr propagated outside of fcn call & it always does its job
     - e.g. vector::size gives nothrow guarantee & always returns size of vector w/o fail
- e.g.

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

  - ↳ if A::q & B::h both provide strong guarantees, f only satisfies **basic guarantee**
    - if a.q() throws, then it provides strong guarantee & it's as if it was nvr called
      - → when expn propagates from f, f hasn't done anything
    - if b.h() throws, it also undos its own effect, but it has no knowledge a.q() ran just prior
      - → when expn propagates from f, effects of a.q() still remain in place
    - program is invalid yet unspecified state, so it gives basic guarantee
  - ↳ maybe effects of a.q() that are impossible to undo
    - e.g. writing to file, printing to screen, & sending network request
  - ↳ for simplicity, assume A::q & B::h only have **local side effects** (i.e. only a & b objs & any associated mem may change)
- e.g. use **copy/swap** to work on temp objs rather than real ones

```
class C {
  A a;
  B b;
public:
  void f(){
    A aTemp{a};
    B bTemp{b};
    aTemp.q();
    bTemp.h();
    a = aTemp;      //copy asgnt operator (could throw)
```

```
    b = bTemp;
  }
};
```
↳ if aTemp.q() or bTemp.h() throw, no changes are made to C obj ∴ it's as if f was nvr called
↳ still **basic guarantee** b/c if b = bTemp throws, we propagate an expn having modified a
↳ need non-throwing swap / asgnt

- one soln is **pImpl (ptr to implementation) idiom**
  ↳ assume ptrs nvr throw
  ↳ e.g.

```
// Impl class has fields
struct CImpl {
  A a;
  B b;
};
class C {
  unique_ptr<CImpl> pImpl;
public:
  void f() {
    unique_ptr<CImpl> temp = make_unique<CImpl>(*pImpl);
    temp->a.q();
    temp->b.h();
    std::swap(temp, pImpl);   // guaranteed nothrow
  }
};
```

  - strong guarantee b/c if a.q() or b.h() throw, all work is done on temp
    → swap will always succeed so f will do its job

- pImpl idiom can also be used in some scenarios for **reducing compilation dependencies**
  ↳ e.g.

```
// w/o pImpl idiom
// C.h
class C {
  A a;      <------ if any of private fields are changed, anybody who #includes "C.h" must recompile
  B b;
  ...
};
```

```
// w/ pImpl idiom
// CImpl.h
struct CImpl {
  A a;
  B b;
};
// C.h
class CImpl;
class C {
  unique_ptr<CImpl> pImpl;
  ...
};
```

  - if CImpl changes, only C.cc must recompile
  - since we're using fwd declaration in C.h, no recompilation required

- can also make CImpl a polymorphic class
  ↳ **Bridge pattern**: swap out implementations at runtime

- how does **vector::emplace_back** give us strong guarantee?
  ↳ easy case is no resizing so we can just put obj in arr
  ↳ when **resizing**:

1) allocate new, larger arr
2) invoke T copy ctor to copy objs of type T to new arr
   → if copy ctor throws, delete new arr & old arr is left intact
3) if no throws, delete old arr & set arr ptr to new arr (nothrow)

↳ complaint for resizing is too many copies so it's better to **move**:

1) allocate new arr
2) move objs from old arr to new arr
   → if move throws, move all objs from new arr to old arr & delete new arr
3) if no throws, delete old arr & set ptr to new arr

- problem is that if move throws once, it might also when moving objs back
  → once we've modified old arr, no guarantee we can restore it
- soln is to declare move ctor as "noexcept" so that emplace_back will perform moves
  → otherwise, it'll copy over every item & do this try-catch block to delete new items if necessary

- if possible, **moves & swaps** should provide **nothrow guarantee**
  ↳ should make this explicit to compiler via **noexcept tag**
  ↳ e.g.

```
class MyClass {
public:
    MyClass(MyClass&& other) noexcept {...}
    MyClass& operator=(MyClass&& other) noexcept {...}
};
```

- if we know fcn will nvr propagate an expn, declare it as noexcept to facilitate optimization
  ↳ moves & swaps should be noexcept at min

LECTURE 23

MULTITHREADING

- **thread** is sequence of ins to be executed
  - ↳ typically just program
- most modern CPUs have more than 1 core so this can sometimes allow extraction of extra efficiency
  - ↳ e.g. **matrix multiplication**

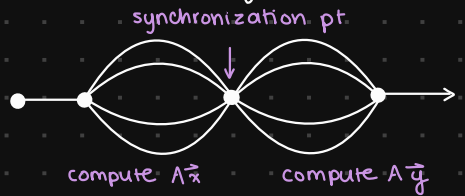
$$\begin{bmatrix} v_1^T \\ v_2^T \\ v_3^T \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \vec{v}_1 \cdot \vec{x} \\ \vec{v}_2 \cdot \vec{x} \\ \vec{v}_3 \cdot \vec{x} \end{bmatrix}$$

$A \qquad \vec{x} \qquad \vec{y}$

  - typically compute entries in order: $\vec{v}_1 \cdot \vec{x}$, $\vec{v}_2 \cdot \vec{x}$, $\vec{v}_3 \cdot \vec{x}$
- in **multithreaded programs**, create separate threads to perform computations simultaneously
  - ↳ e.g.

```
// allows us to multiply each row for our product Ax
void multiplyRow(int row, const vector<vector<int>>& A, const vector<int>& x, vector<int>&
                 output){
    output[row] = 0;
    for (int i = 0; i< A[row].size(); ++i){
        output[row] += x[i] . A[row][i];
    }
}
vector<vector<int>> A{{1,2,3}, {4,5,6}, {7,8,9}};
vector<int> X{10, 11, 12};
vector<int> Y{0, 0, 0};
vector<thread*> threads;
for (int i=0; i< A.size(); ++i){
    threads.push_back(new thread{multiplyRow, i, std::cref(A), std::cref(X), std::cref(Y)});
}
for (thread* t: threads){
    t->join();      // waits & stalls program until it's finished running
    delete t;
}
// by this pt, all threads have been completed & we can now use Y
```

  - why do we need ref / cref if thread ctor implicitly copies / moves its params?
    - → **cref & ref** create ref-wrapper obj types, which can be copied / moved
  - distributed workload & each thread works independently on separate part of problem
  - example of embarassingly parallelizable problem
- e.g. **sychronization**: $A^2\vec{x}$
  - ↳ first compute $A\vec{x} = \vec{y}$ & then compute $A\vec{y} = A^2\vec{x}$
  - ↳ need to compute $\vec{y}$ in its entirety before computing $A\vec{y}$
  -



  - spawn threads for $A\vec{x}$ & join them, then spawn threads for $A\vec{y}$ & join them
    - → creating threads is expensive so better to **reuse same threads**
- e.g. **barrier**: 2 fcns, f1 & f2, where both must complete phase 1 before they can proceed to phase 2

```
void f1(){
    cout << "Thread 1 Phase 1" << endl;
    t1Ready = true;
```

```
    while (!t2Ready) {}
    cout << "Thread 1 Phase 2" << endl;
}
void f2() {
    cout << "Thread 2 Phase 1" endl;
    t2Ready = true;
    while (!t1Ready) {}
    cout << "Thread 2 Phase 2" endl;
}
```

↳ use \n instead of endl for better line placement

- use `std::atomic<bool>` to prevent incorrect compiler optimizations w/ multithreaded code