CS 247 Notes

January 13, 2023

Benjamin Chen

1 May 3rd

Rob Hackman
DC 2110
Office Hour: Tuesdays 4-5
In this class, we will be learning about OOP (Object-Oriented Programming) through the lenses of C++. We make abstract data types (ADTs) to model components of our systems of the real world. An ADT should provide an interface to the clientthat allows for the operations necessary, without knowing the implementation.
For example, a class to represent rational numbers.

```
class Rational { // could have used the struct
                 // keyword, only difference is
                 // default visibility, struct
                 // is public by default,
                 // class is private
    int num;
    int den;
public:
    Rational(int num, int den): num{num}, den{den} // Member of initialization list (MIL)
    {} // constructor body
};

struct Rational {
    private:
        int num;
        int den;
};
```

Steps of object creation are:

1. Space is allocated
2. Fields are initialized/constructed (MIL happens here)
3. Constructor body runs

The following code won't link with the above Rational class.

```
class RatPair {
    Rational r1;
    Rational r2;

public:
    RatPair (int n1, int d1, int n2, int d2) {
        // by this point r1 and r2 must be initialized
        // but there is no default ctor
        r1 = Rational {n1, d1};
        r2 = Rational {n2, d2};
    }
};

int main() {
    RatPair r; // Asks for a RatPair on the stacks
}
```

Compiler allocates space on the stack constructs the object following the 3 steps above. At step 2, it must initialize fields r1 and r2. If you don't tell the compiler how to do this, it assumes default construction. BUT, Rational has no default constructor.
The solution is to provide default constructor in Rational, OR, tell the compiler how to construct these fields.

```
class RatPair {
    Rational r1;
    Rational r2;

public:
    RatPair (int n1, int d1, int n2, int d2) : r1{n1,d1}, r2{n2,d2} {}
}
```

MIL tells compiler how to initialize the fields before ctor body runs
MIL must be used for const fields and references.

```
class Student {
    string name;
    const int SID = 12345678;
};
```

This would make every student to have the SID 12345678.

```
class Student {
    string name;
    const int SID;
public:
    Student (string name, int SID) : name{name}, SID{SID} {}
};
```

Now, to make a good rational class.

```
class Rational {}
    int num;
    int den;
public:
```

```
    Rational(int num, int den): num{num}, den{den} {}
    int Num() {return num;}
    void Num(int n) {num = n;}
    int Den() {return den;}
    void Den(int d) {den = d;}
};
```

Now, let's take a look at the linked lists.

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

// Creates a list from 1 to n
Node * createList(unsigned int n) {
    Node *np = new Node{n, nullptr};
    while (--n) {
        np = new Node {n, np};
    }

    return np;
}

int main() {
    Node *p = createList(10);
    for (Node *i = p; i != nullptr; i = i -> next) {
        cout << i->data << endl;
    }
}
```

Now, the program compiles but it leaks memory. We should implement a destructor to fix the memory leak.

```
class Node {
public:
    Node * next;
    int data;
    Node (int d; Node * next): next{next}, data{d} {}
    ~Node() {
        delete next;
    }
};

int main() {
    Node *p = createList(10);
    for (Node *i = p; i != nullptr; i = i -> next) {
        cout << i->data << endl;
    }
    delete p;
}
```

What happens if I do:

```
int main() {
    Node *p = createList(10);
    for (Node *i = p; i != nullptr; i = i -> next) {
        cout << i->data << endl;
    }
    Node n = *p;
    delete p;
}
```

Then the copy `n` has a `next` already deleted.

2 May 5th

n's tail is gone. Accessing the rest of n's list is a memory error when n gets destroyed.
Double free error.
Every class, unless otherwise stated by the programmer, has two additional important functions.

1. Copy constructor - used when constructing an object with another of the same type

   The compiler provides a built-in copy ctor that just copy initializes all fields (e.g. bitwise copies Plain-old-data (POD), and copy constructs fields that are objects)

   In the case of Node, this just copies the int for data, and the points. So we get a shared tail.

   This is a shallow copy, we would like a deep copy where each list has its own distinct tail.

   The compiler provided copy constructor is insufficient, so we must provide our own.

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

   Node n = *p; // copy ctor
   n = *q;
   // already exists, not a construction;
   // = here is the assignment operator
   // For objects, we call it the copy assignment operator
   ```

   As with the copy ctor, the compiler provides a built-in copy assignment operator that just copy assigns all fields.

   So, we'd like to override this behavious.

2.

Aside - References

```
int n = 10;
int &r = n;
r = 50;
cout << &n << "," << &r << endl;
// Same address!
int q = 10;
```

```
int *p = &q;
*p = 50;
cout << &q << "," << &p << endl;
// Different address!
```

References

- Are not guaranteed to have any more memory
- Can't be reseated (changed to refer to something else)
- Must be initialized (No such thing as a null reference)
- Cannot declare an array of references

Operator overloading

giving meaning to built-in C++ operators for types they are not already defined for.

```
struct Node { ... };

Node operator+(const Node & lhs, const Node & rhs) {
    Node head { lhs.data + rhs.data, nullptr}
    Node * curr = &head;
    for (Node * l = lhs.next, *r = rhs.next; l || r ;
    (l ? l = l->next: nullptr, r ? r = r->next : nullptr)) {
        Node * next = new Node { (l? l->data: 0) + (r ? r->data : 0), nullptr};
        cur->next = next;
        cur = next;
    }
}

Node * n = createList(5); // 1 -> 2 -> 3 -> 4 -> 5
Node * p = createList(3); // 1 -> 2 -> 3
Node z = *n + *p;
// z 2 -> 4 -> 6 -> 4 -> 5

delete n;
delete p;
```

Adding two lists

```
Node getmeANode (int n, int p) {
    Node * next = new Node {p, nullptr};
    Node q{n, next};
    return q;
}
```

3 May 10th

We can overload operators, it is just like defining any other function. The only difference is the function name is `opreator` followed by the actual operators you want to overload.
e.g.

```
Node operator+(const Node & lhs, const Node & rhs) {
    ...
}
```

However, we are going to want our fields to be private. Adding two nodes together is part of the interface for our class we'd like to offer - so operator+ should be a member.

```
struct Node {
    ...
    Node operator+(const Node & rhs) {
        // first operand is implicitly the Node
        // pointed to by this
        // function is as before, lhs and rhs get
        // replaced with *this and this-> respectively
    }

    // Now the copy assignment operator
    // Motivation:
    // x = y = z <==> x = (y = z)
    // for this to do what we want, assignment must return a value,
    // we use the object that was assigned by convention and,
    // by necessity later
    Node & operator=(const Node & other) {
        if (&other == this) return *this;
        delete next;
        data = other.data;
        next = other.next ? new Node{*other.next} : nullptr
        return *this;
    }
}
```

But `new` can fail, if it does, the function ends and our node now contains a dangling pointer. How to fix it?

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

Or, simply

```
Node & operator=(const Node & other) {
    Node tmp{other};
    delete next;
    next = tmp.next;
    tmp.next = nullptr;
    data = tmp.data;
    return *this
}
```

Or, use the tmp Node to do the cleaning.

```
#include <utility>

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

    Node & operator=(const Node & o) {
        Node tmp{o};
        swap(tmp);
        return *this;
    }
}
```

In this way, old data are freed by `tmp`. This is the copy-and-swap idiom.
Now, consider:

```
Node * p = createList(10000);
Node * q = createList(20000);
Node z = *p + * q;
```

Operator+ gets called, constructs its local list, that list gets returned. It is copied out to the callers stack frame, as a temporary object; that temporary object is then copied into z.
Why are we wasting all this time deep copying two lists that we're only going to delete moments later?
The answer to this problem, is to understand another C++ type.

```
int & x = 4 + 5; // illegal
```

4 + 5 is the temporary return value of the addition operator adding 4 and 5. We can refer to it this way, as it ceases to exist once the statement `int & x = 4 + 5;` is done.
This is an r-value, r-values are temporaries that don't exist past current statement they exist in.
When a function returns, its stack allocated variables all become rvalues, as they don't exist past the end of the statement.
Also, return values such as $4 + 5$ are rvalues.
One more rule about references…
You cannot have a reference to a reference.
e.g. int && x;
In fact, this means something else.

```
int & x ...  // an l-value reference
int && x ... // an r-value reference
```

l-values are persistent values that have some lifetime beyond the current statement, have their own memory associated with they.

4 May 12th

l-values can be assigned, unless they are const.
Consider:

```
int foo(int &x) {
    x += 1;
    return x;
}

int main() {
    foo(5);
    // Not valid
}
```

But, if we consider the following:

```
int bar(int &y) {
    return y + 1;
}

int main() {
    // Still invalid
    cout << bar(5) << endl;
}
```

Instead,

```
int bar (const int &y) {
    return y + 1;
}
int main() {
    cout << bar(5) << endl;
    // valid and compiles
    // compiler will make a temporary
    // memory location for 5 if it didn't
    // already have one and let y refer to that
}
```

int & is an l-value reference (a reference to an l-value) - if a parameter is a const l-value reference, we can validly accept r-values as arguments.
int && is a r-value reference (a ref to an r-value), only r-values args.

```
// x and y are nodes that are lists
Node z = x + y;
// the list from operator+'s stack frame is
// copied out to the caller's stack, to a
// temporary, that temporary is then copied into
// z, two copies of two lists that are just going to be
// destroyed anyways.
```

It is wasteful to spend all this time copying something thats going to be destroyed. Anyways, so, we can define a new ctor and another overload for the assignment operator that operate on r-values instead, respectively called te move ctor and the move assignment operator.

```
struct Node {
    // as before
    Node (Node && o) : data{o.data}, next{o.next} {
        o.next = nullptr;
```

```
    } // move constructor
    Node& operator=(Node &&o) {
        delete next;
        next = o.next;
        o.next = nullptr;
        data = o.data;
        return *this;
    }
}
```

Instead ...:

```
Node & operator=(Node && o) {
    swap(o);
    return *this;
} // when r-value o is destroyed
// it will free our old data
```

Note: in cases like ours e.g

```
// x and y are Nodes
Node n = x + y;

// The compiler is allowed to elide these ctors, opting to instead build the return value
```

It is allowed to do so even if the ctor has side effects.
Our class is terrible! It has no encapsulation.
Our classes have invariants. Invariant is a statement that must always hold true. Without being able to assume invariants, code can become very impossible to write. In fact, we used an invariant in our class, but because we have no encapsulation, we have no guarantee that it actually holds!
We assumed next is always either a valid heap allocated node, or the nullptr.
But because we're porrly encapsulated...

```
int main() {
    Node a {3, nullptr};
    Node b {2, &a};
    Node c {1, &b};
}
```

Our invariant no longer holds! When b and c goes out of scope, they will try to call delete on portions of the stack. That is not valid!

5 May 17th

Our `Node` class relied on the invariant

- The `next` pointer is always either
  - A valid heap allocated node
  - or, the `nullptr`

This is necessary, because our destructor is

```
Node::~Node() {
    delete next;
}
```

The problem is that we cannot maintain this invariant.
For example,

```
Node a {3, nullptr};
Node b {2, &a};
```

b's destructor tries to delete a, but a is on the stack. Error!
The solution to this problem is to properly encapsulate our class so that the client can't violate our invariants.
Recall: an encapsulated class should be able treated like a black box, an ADT where all the client cares about is the abstract data type. (in this case a list) NOT the implementation. (in this case a linked list).
We can make a new class, List, that the client uses, that encapsulates the implementation. Our node class already has the big-5. (Copy constructor, CAO, move ctor, MAO, dtor).
So, our List class will use that.

```
class List {
    struct Node; // private nested class
    Node * head;
public:
    List() : head{nullptr} {}
    List(const List & o) : head{o.head ?
            new Node{*o.head} : nullptr} {}
    List & operator=(const List &o) {
        *head = *o.head;
        return *this;
    }

    List(List && o) : head{o.head} {
        o.head = nullptr;
    }

    List & operator=(List && o) {
        *head = std::move(*o.head);
        // std::move is a function in the <utility> header
        // it forces a value to be treated like a r-value
        return *this;
    }

    ~List () {delete head;}

    void cons(int n) {
        head = new Node {n, head};
    }
}
```

Great, our client can create lists but how do they use them? Well, they need to be able to access each element, so we must provide a mean to do so.

```
class  List {
    // as before
    int & ith (int i) { // pre-condition: i is in range
        Node * p = head;
        while (i && head) {
            i--;
```

```
            head = head->next;
            return head->data;
            // assume i was in range.
        }
    }
}

int main() {
    List l();
    l.cons(3); l.cons(2); l.cons(1);

    for (int i - 0; i < 3; ++i) {
        ++(l.ith(i));
        cout << l.ith(i) << endl;
    } // works ... but
}
```

But, now looping over the list is $O(n^2)$, because `ith` is $O(n)$. We can't just return nodes to the client, that breaks our encapsulation. We can instead, return an object that encapsulates the idea of a pointer to our list.
Our list - An iterator

```
class List {
    public:
        class Iterator {
            Node * p;
            public:
                int & operator*() {
                    return p->data;
                }
                Iterator & operator++() {
                    if (p) p = p->next;
                    return *this;
                }

        }
}
```

Note: to implement postfix operator, ++, You must include a second token int parameter

```
Iterator operator++(int) {
    Iterator it{p};
    if (p) p = p->next;
    return it;
}
Iterator(Node *p) : p {p} {}

bool operator!= (const Iterator &o) {
    return p!= o.p;
}


Iterator begin() {
```

```
    return Iterator{head};
}

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

int main() {
    List l;
    l.cons(3); l.cons(2); l.cons(1);

    for (List::Iterator it = l.begin(); it != l.end(); ++it) {
        cout << *it << endl;
    }
}
```

Saying `List::iterator` is cumbersome, the compiler knows what type `List::begin` returns, so instead why don't we tell the compiler to figure out the type?
So instead,

```
for (auto it = l.begin(); it = l.end(); ++it) {
    cout << *it << endl;
}
```

`auto` asks the compiler to deduce types based on the initialization of a variable. It is only possible when compiler can definitively determine the type. But, you shouldn't use `auto` all the time. It reduces readability, but it is fine for iterators.
This is the iterator design pattern, it solves the problem of "how do I allow iteration over my container class efficiently without breaking the encapsulation".
This is the essence of the design pattern - a design pattern is if you have a problem like x, then maybe this is a good solution.
The iterator pattern is so pervasive to C++ programming that there is even specialized syntax for it.
e.g.

```
for (auto it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl;
}
\\ OR
for (auto x : l) { // range-based for loop
    // x is an int (or whatever the container contains)
    cout << x << endl;
}
```

Caveat:

```
List l; l.cons(3); l.cons(2); l.cons(1);

for (auto x: l) {++x}
// the variable x is a local copy of the value returned by
// dereferencing each iterator, can be solved with ...
for (auto x: l) { cout << x << endl;}
// prints 1, 2, 3
```

6 May 19th

```
for (auto & x : l) {
    // only if operator& for
    // the iterator returns a ref
    ++x;
}

for (auto x: l) { // & or not does not matter, no mutation here
    cout << x << endl; // prints 2, 3, 4
}
```

The `for(... : ...)` is called a range based for loop, and it can be used IF `for(x : C)`, `C` is a type that defines functions `begin()` and `end()` which return a type `Q`.
Type `Q` must have been defined

- prefix increment operator
- dereference operator
- inequality operator

Minor encapsulation issue, Iterator's ctor is public, client can construct their own iterators, but should only be able to through `begin` and `end` - solution is to make the constructor private.

```
class List {
    public:
        class Iterator {
            Iterator(Node & cur): ... {}
            ...
            public:
            ...
                friend class List;
                // can go anywhere in class iterator
        };
};
```

But now, `List` can't access the constructor. The solution is to declare `List` a friend.
This means `List` can access Iterator's private members, but not vice versa.
In general, try to havec as few friends as possible. It weakens encapsulation.
You can also declare functions as friends. This is particularly useful for two operators that should not be member functions.
Recall: when operators are overloaded as member functions, the first operand is always the object pointed to by this. We don't always want the first operand to be the type of our object.
e.g.

```
class Pair {
    int x, y;
    public:
        Pair(int x, int y) : x{x}, y{y} {}
        istream & operator>>(istream & in) {
            in >> x >> y;
        }
        ostream & operator<<(ostream & out) {
            out << "(" << x << "," << y << ")";
        }
};
```

These operators are member functions, the first operand is then a `Pair`. So:

```
Pair p{1, 2};
cout << p << endl; // Wrong!
cin >> p; // Wrong!
```

Our function is defined as

```
Pair p{1, 2};
(p << cout) << endl;
p >> cin;
// Works, but weird.
```

Solution is to declare as friends.

```
class Pair {
    ...
    public:
    friend istream & operator>>(istream &, Pair &);
    // operator<< similar
};

// Outside class
istream & operator>>(istream & in, Pair & p) {
    return in >> p.x >> p.y;
}
```

Separate Compilation

```
// student.h
#include <string>

class Student {
        const int ID;
        std::string name;
    public:
        Student(int id, std::string name);
};

// course.h
#include "student.h"
class Course {
        int numStudents;
        Student * students;
        int enrolled;
    public:
        Course(int numStudents);
        void enroll(const Student &s);
        ...
};

// student.cc
#include <string.h>
#include "student.h"
```

```
using namespace std;
Student::Student(int id, string name): id{id}, name{name} {}

// course.cc
#include <string>
#include "student.h"
#include "course.h"
Course::Course(int n) : numStudents{n}, students{new Students*[numStudents]}, enrolled{0} {}
void Course::enroll(const student &s) {
    students[enrolled] = new Student{s};
    ++enrolled;
}

// main.cc
int main() {
    Course cs247{140};
    Student s1{0, "Albert"};
    cs247.enroll(s1);
}
```

How to compile? Well, we can do `g++ -std=c++14 student.cc course.cc main.cc -o main`
Compiles and links all three files, creates executable.
BUT, if I make a change to `student.cc`, now to rebuild my program. I must recompile all files! Wasteful, compilation can take a long time.
Better, compile files separately into object files (compiled binary files that are not complete programs). Then, link object files together into the final executable.
e.g. `g++ -std=c++14 main.cc` The `-c` flag means compile only, don't link. It will create `main.o` here, an object file.

```
g++ -std=c++14 -c student.cc
g++ -std=c++14 -c course.cc
g++ main.o student.o course.o
```

If we change a file only need to recompile files that depend on it, e.g, if we change `student.cc`, we only need to

```
g++ -std=c++14 -c student.cc
g++ main.o student.o course.o
```

e.g. If I change `student.h`, we need:

```
g++ -std=c++14 -c main.cc
... student.cc
... course.cc
```

These all depend on `student.h`

```
g++ main.o student.o course.o -o main
```

7 May 24th

```
// Vec.h
#include <iostream>
// headers never use namespace
class Vec {
    int x, y;
    public:
        Vec(int x = 0, int y = 0); // Default parameters
                                   // only need to be in header
```

```
        friend std::ostream & opreator<<(std::ostream &, const Vec &);
};

// Vec.cc
#include "Vec.h"
#include <iostream>
using namespace std;
// If parameters are defaulted, this function just
// gets called with the default values
Vec::Vec(int x, int y) : x{x}, y{y} {}
ostream & operator<<(ostream & out, const Vec &v) {
    out << v.x << "," << v.y;
    return out;
}
```

Note on defaulted function parameters - they must be the last n parameters.
e.g

```
int foo(int x = 0, int y, int z = 1) {
    return (x + y) / z; // Invalid!
                        // Defaulted 2 parameters are not the last 2 parameters
}

foo(3, 5);
```

This is ambiguous! is this $x = 3, y = 5, z = 1$? Or is it $x = 0, y = 3, z = 5$?

```
// Basis.h
#include "Vec.h"
class Basis {
    Vec v1, v2;
    public:
        Basis(const Vec &, const Vec &);
};

// Basis.cc
#include "Basis.h"
#include "Vec.h"

Basis::Basis(const Vec & v1, const Vec & v2) : v1{v1}, v2{v2} {}

// main.cc
#include "Vec.h"
#include "Basis.h"
#include <iostream>

using namespace std;
int main() {
    Vec v1{1, 0};
    Vec v2{0, 1};
    cout << v1 << " " << v2 << endl;
    Basis b{v1, v2};
}
```

`g++ -c Vec.cc`
`g++ -c Basis.cc` // Breaks
`g++ -c main.cc` // Breaks
You can declare things as many times as you want, but you can define only once.
The problem is that `struct Vec` is being defined twice. `Basis.cc` and `main.cc` include `Vec.h` and `Basis.h`, but `Basis.h` includes `Vec.h`. So, `main.cc` and `Basis.cc` get 2 copies of the contents of the `Vec.h` - the `#include` directive (a preprocessor directive) copies and pastes the included header file right there.
In order to stop a file getting included twice, we must intervene at the preprocessing.
Preprocessor transforms your program before the compiler even sees it.
Two useful preprocessor directives
`#define FLAGNAME`
-defines the preprocessor variable `FLAGNAME` to exist.
If `FLAGNAME` is not defined, everything within the `if` stays in your program (iif not defined).
If it is defined, everything contained is removed before compiler sees it. `#ifndef FLAGNAME`
`#endif`

```
// new Vec.h
#ifndef VEC_H_
#define VEC_H_
    {
        // As before
    }
#endif
```

This is called a header guard, use on all headers to avoid Double Inclusion.
Now, we compile and link `g++ -std=c++14 -c Vec.cc`
`g++ -std=c++14 -c Basis.cc`
`g++ -std=c++14 -c main.cc`
`g++ main.o Vec.o Basis.o -o main`
If I change `Vec.cc`, what must I do?
Only need to recompile `Vec.cc` and relink.
If I change `Basis.h`, what must I do?
Recompile `Basis.cc`, `main.cc`, and relink.
I don't want to keey track manually each time of what files have changed, and what files I must recompile because of that.
So, the solution is to: Unix Makefiles.

Makefile

Example - Makefile

```
main : main.o Vec.o Basis.o
\t  (must be a tab character, not spaces)
    g++ main.o Vec.o Basis.o -o main
main.o : main.cc Basis.h Vec.h
    g++ -std=c++14 -c main.cc
Basis.o : Basis.cc vec.h basis.h
    g++ -std=c++14 -c Basis.cc
Vec.o : Vec.cc Vec.h
    g++ -std=c++14 -c Vec.cc
```

A more general Makefile.

```
CXX = g++
CXXFLAGS = -std=c++14 -Wall -MMD
EXEC = myprogram
OBJECTS = main.o vec.o basis.o
DEPENDS = ${OBJECTS:.o=.d}

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-#include ${DEPENDS}

.PHONY: clean

clean:
    rm ${OBJECTS} ${EXEC} ${DEPENDS}
```

Consider the following examples:

```
// Student.h
#include "course.h"

class Student {
    const int ID;
    std::string name;
    Course *courses; // courses we are in, capped at 5

};

// Course.h
#include "Student.h"

class Course {
    std::string name;
    Student ** enrolled;
    const int capacity;
    int currentlyEnrolled;
    public:
        Course(std::string, int capacity);
        void enroll(Student *);
};
```

8 May 26th

```
// course.cc
#include "course.h"
#include "student.h"

Course::Course(string name, int maxSize) :
        capacity{MaxSize}, name{name}, enrolled{new Student * [capacity]} currentlyEnrolled{
```

This is a bug.
Regardless of MIL order, fields are initialized in the order they were declared in the type. So `enrolled`'s initialization occurs before capacity's.

So for `new Student *[capacity]`, we don't know what `capacity` is!
Solutions:

- initialize enrolled as `new Student * [maxSize]`
- OR, switch order of declaration in class

The compiler will warn you if your MIL order doesn't match the declared order.

Static keyword

We could use `static const int maxCourses = 5`.
Static says this variable belongs to the CLASS, not an individual object.

Forward Declaration

Problem - We try to compile `student.cc` (or `course.cc` or `main.cc`) and we get an error, `student.cc` includes `course.h` and `student.h`. `student.h` includes `course.h`, and `course.h` includes `student.h`. That is a cyclical include.
But, in `student.h`, the header guard is defined. So when `student.h` includes `course.h`, the content of `course.h` are pasted into `student.h`. But, since `student.h` header guard variable is already defined, `course.h`'s include of `student.h` comes back empty. So, the pasted copy of `course.h` refers to type `Student`, but doesn't know where it exist yet.
But, do our headers really need to include each other? Consider:

```
//Student.h
class Student {
    const int ID;
    std::string name;
    Course ** courses;
    public:
        ...
};
```

The size of `Student` replies only on the size of an `int`, a `string`, and a pointer. We don't need to know here, the size of a course.
There is no true compilation dependency between `student.h` and `course.h`. We only need to know that a type named `Course` exists.
Solution is to forward declare the class. Don't include the header.

Compilation Dependency

A compilation dependency exists if you must know the size of types in that header, or details of existing functions. Don't introduce compilation dependencies with includes where none exist!

```
// a.h

class B; // forward declaration

class A {
    int x;
    public:
        void Foo(B b);
};
```

```
// b.h
#include "c.h" // include
// need to know size of C to know size of B, must include
class B {
    class B {
        C c;
        public:
            ...
    }
};

// c.h
class A; // forward declaration
class C {
    A * ap;
    public:
        ...
}

// d.h
#include "e.h" // include
class D {
    ...
    public:
    void Bar(E * e) {
        e->baz();
    }
}
```

Polymorphism

There are two types of students, coop and non-coop students. But they are both students. i.e. We don't want 2 arrays in our course, one for coop students, one for regular.
How to achieve this? What we want is polymorphism (many forms) through one generalized type (on interface) we get many specialized behaviours.

```
class Student {
    ... // as before
    public:
        int Fees();
        ...
};

class CoopStudent : public Student { // is-a relationship
    // inheritance
    int Fees()
}
```

9 May 31st

```
// Student.cc
int Student::Fees() {
```

```
    return 2500 + 500 * enrolledCourses;
}

// CoopStudent.cc
int CoopStudent::Fees() {
    return 4500 + 500 * enrolledCourses;
}
```

We try to compile this, it doesn't work. `enrolledCourses` is private in `Student`. So, only `Student` methods and `friend`s can access it! We must give `CoopStudent` access somehow.

protected keyword

`protected` means accessible in this class and derived classes.
We can add:

```
    protected:
        int getNumCourses();
```

Note: Member-Initialization-List can also call functions.
We can't initialize `ID` and `name` directly in `CoopStudent` ctor.

1. They are private in `Student`
2. By the time we initialize fields, they've already been initialized.

Phases of object initialization

1. Space is allocated
2. Superclass components are initialized
3. Fields are initialized
4. Ctor body runs

Phases of object destruction

1. Destructor body runs
2. Fields are destroyed.
3. Superclass component is destroyed
4. Space is deallocated

Also since `Student` has no default constructor, we must specify how to initialize it in the MIL, as the compiler doesn't know how to initialize it otherwise.

```
class Course {
    ...
    public:
        int enrolledFees() {
            int x = 0;
            for (int i = 0; i < curEnrolled; ++i) {
                x += enrolled[i]->fees();
            }
            return x;
        }
}
```

`Course`'s `enrolled` array is an array of `Student` pointers, but we stuck a pointer to a CoopStudent in there - how? That's the beauty of public inheritence. A `CoopStudent` is-a `Student`, so a `Student *` can point at a `CoopStudent`.
So, we can work polymorphically on different kinds of `Students` through `Student` pointer's except that's not what we got, we got just the behaviour of `Student`.
The compiler chooses which function (fees) to run based on the `static` type of the data, so if I, say:

```
cout << a.fees() << endl;
```

I get 5000, because the compiler sees `a`'s type is `CoopStudent`, and so it calls `CoopStudent::Fees`
But in `Course::enrolledFees`, we are operating on `Student` pointers. So it calls `Student::Fees` regardless of what type the pointer actually points at.
How do we solve this so the compiler chooses the method based on the actual run-time type of an object, not the static type.
Solution - declare the method `virtual`
All of a sudden, we get the behaviour, we want, the Fees called on `Student` pointers calls the appropriate method based on the run-time type of the object. But how is this possible?

```
// before fees was virtual
cout << sizeof(Student) << " " << sizeof(CoopStudent);
// printed 48 48
// after fees was virtual
cout << sizeof(Student) << " " << sizeof(CoopStudent);
// printed 56 56!
```

So, the addition of a virtual function made our object larger by 8 bytes. (The size of a pointer)
Quick Note: The C++ standard only specifies that compiler must implement polymorphic behaviour of virtual functions through base class pointers and references. Not how the compiler must do it. However, virtually all compilers implement it this way.
Why are our objects a pointer larger? THe compiler knows it may be asked to call this virtual function through a base class pointer or reference and it must figure out which version to call.
So, everytime the compiler creates an object of a class with a virtual method, it sticks an extra pointer in there.

10 June 2nd

The inclusion of virtual functions made our object 8 bytes (the size of a pointer) larger. That is because at the first 8 bytes of these objects, the compiler is storing a pointer called the virtual table pointer or vptr. It points at the virtual function (or vtable) for the class of that object's type.
e.g.

```
class A {
    int x;
    public:
    virtual void hello() {
        cout << "I'm an A";
    }
};

class B {
    int y;
    public:
    void hello() {
```

```
        cout << "I'm a b";
    }
}

int main() {
    A a1{5};
    A a2{3};
    B b1{1, 2};
    B b2{7, 8};
}
```

WARNING - never use arrays of objects polymorphically.

```
class One {
    int a, b;
    public:
        One(int a, int b) : a{a}, b{b};
};

class Two : public One {
    int c;
    public:
        Two(int a, int b, int c) : One{a, b}, c{c} {}
};

void foo(One arr[]) {
    arr[0] = {7, 8};
    arr[1] = {9, 10};
}

int main() {
    Two arr[2] = {{1, 2, 3}, {4, 5, 6}};

    foo(arr);
}
```

Override specifier

```
class Book {
    int pages;
    string title, author;
    public:
        Book (int pages, string title, string author) :
            pages{pages}, title{title}, author{author} {}
        virtual bool isHeavy() { return pages > 200; }
};

class Comic : public Book {
    string hero;
    public:
        Comic (int pages, string title, string authour, string hero):
            Book(pages, title, author), hero{hero} {};
    bool isHeavy() override { return pages > 50; }
};
```

Quick Note - the override keyword does not change the behaviour of your program. It just asks the compiler to check "if this is a valid override". That is, check that if there is a matching virtual function exists in a parent class.

```
class Text : public Book {
    string topic;
    public:
        Text(int pages, string title, string author, string topic):
            Book{pages, title, author}, topic{topic} {}
        bool isHeavy() override { return pages > 500; }
}

int main() {
    Book ** library = new Book *[3];
    library[0] = new Book{453, "Spin", "Robert Charles Wilson"};
    library[1] = new Comic{57, "Spider-man does a thing!", "Stanley", "Morbius"};
    library[2] = new Text{5373, "C++ - the language", "Bjarne Stroustrup", "C++"};
}

void isLibraryHeavy(Book ** library, int size) {
    for (int i = 0; i < size; ++i) {
        cout << library[i]->getTitle() << " is heavy?" << library[i]->isHeavy() << endl;
    }
}
```

This works, but however, if we consider the following:

```
void copyFirstBook(Book ** lib, int size) {
    for (int i = 0; i < size; ++i) {
        *lib[i] = *lib[0]
    }
}
```

This invokes the copy assignment operator, but its not virtual. So this calls `Book`'s assignment operator - only assigns the `Book` components of these objects. (i.e. `pages`, `title` and `author`). This is partial assignment.

11 June 7th

```
class Book {
    string title, author, field;
    public:
        ...
        Book & operator= (const Book & o);
}

class Comic : public Book {
    string hero;
    public:
        Comic & operator=(const Comic & other) {
            Book::operator=(other);
            hero = other.hero;
        }
```

```
}

// text is similar
void copyFirst(Book ** lib, int size) {
    for (int i = 1; i < size; ++i) {
        *lib[i] = *lib[0];
    }
}
```

So, the CAO for `Book` isn't virtual. So this only calls `Book`'s assignment operator, so each object gets its `Book` fields assigned, but not its `Comic` or `Text` fields if it is one such object - this is a partial assignment. So, we try to make `Book`'s CAO virtual. When we add the `override` keyword to `Comic`'s (and `Text`'s) CAO I get told, you asked to override a function, but no matching virtual function exists in the base class. So, in order to override `Book`'s CAO, `Text` and `Comic`'s CAOs must take `Book &` parameters (types of the functions must match). If I do that, I can't access tbe `hero/topic` field of `other`, because it is a `Book &`. Let's assume that I could write these CAO's, if I can, what happens when I do this?

```
Book * pt = new Text{...};
Book * pt = new Comic{...};
*pt = *pb
```

So if the CAO is `virtual`, then mixed assignment is allowed because `Comics`, `Books`, and `Texts` are all `Books`. So they all match the CAO parameters.
So we get partial assignment through base class pointers if CAO is non-virtual, and mixed assignment if it is. The problem is that assignment through Base class pointers doesn't really make sense.
It doesn't make sense because we never know if those two pointers actually point to the same type.
So the solution is - just don't do it! Disallow assignment through base class pointers.
So, we make the base class CAO protected.

```
class Book {
    ...
    protected:
        Book & operator=(const Book & o);
     public:
        ...
};

class Comic: public Book {
    public:
        Comic & operator=(const Comic & o) {
            Book::operator=(o);
            hero = o.hero;
            return *this;
        }
}


void copyFirst (Book *lib, int size) {
    for (i = 1, i < size; i++) {
        *lib[i] = *lib[0];
        // No longer valid, as we wanted.
    }
}
```

```
Book b{457, "Spin", "Robert CHarles Wilson"};
Book a{320, "Annihilation", "..."};
a = b;
```

The line above is no longer valid, `Book`s CAO is protected.
So preference is that all base classes be abstract. Any abstract class is a class that cannot be instantiated, an abstract class in C++ is any class with at lease one pure virtual method.
A pure virtual method is a method that doesn't require an improvement.

```
class CoopStudent : public Student {
    ...
    public:
        int Fees(return 4500 + sco * numCourses   )
};

class Student {
    public:
        virtual int fees() = 0;
}
```

$= 0$ states this function is a pure virtual. It does not require an implementation. (But still give g one)
Doesn't make sense for `Student::Fees` to have an implementation (all `Students` are either `Regular` or `Coop`) So we don't give it one. Since student has a pure-virtual method if it is on a abstract class.

```
Student S{...};
```

This is not valid. We can't instantiated objects of an abstract type.

```
class Student {
    Course ** arr;
    int numCourses;

    virtual int fees() = 0;
    Student();
    // Provided so our derived classes can
    // instantiate their student components.
}
```

Back to `Book`:

```
class AbstractBook {
    int numpages;
    string title, author;
    protected:
        AbstractBook & operator= (const AbstractBook & other);
    public:
        ...
        virtual ~AbstractBook() = 0;
}
```

~AbstractBook still needs an implementation because the 4 phases of object destruction are:

1. Dtor body runs
2. Dtor for fields which are objects run

3. Base class dtors run
4. Space is deallocated

So if the dtor isn't implemented, we get a linker error - so implement `AbstractBook::~AbstractBook() {}`.

```
class REgularBook : public AbstractBook {
    ...
    public:
        ...
        RegularBook & operator=(const RegularBook &);
        ...
}
```

For `Comic` and `Text`, they are similar.
Now, assignment through base class pointers is no longer allowed, but assignment of `RegularBooks`, `Comics`, and `Texts` is, so we are happy. Note, non-abstract classes are called concrete classes.
If we had no method to make pure virtual, we chose to make the dtor pure virtual. Readon for this si that a class with inheritance should always have a virtual dtor - why?
Consider:

```
class One {
    int * p;
    public:
        One(int n) : p {new int{n}} {};
        ~One() { delete p };
        // Not virtual
}

class Two : public One {
    int *x;
    public:
        Two(int a, int b) : One{a}, x{new, int{b}} {}
        ~Two() { delete x };
}

One *arr[3];
arr[0] = new One{5};
arr[1] = new Two{3, 7};
arr[2] = new Two{4, 5};

for (int i = 0, i < 3; ++i) {
    delete arr[i];
}
```

Since the dtor isn't virtual, virtual dispatch doesn't happen. Only `One`'s dtor is called, we leaked all `Two::x` fields! Always, if you have inheritance, declare your dtor virtual.

System Modelling

- Software systems can grow large and complex
- It's not always feasible that all programmers working on a project know every single module.

- Models can be an effective way of communicating the structure of a software system. We are going to use the UML class diagrams.

How to draw a class:

- class name
- fields (optional)
- methods (optional)

12 June 9th

Whether or not to include fields/methods (or in fact which fields/methods to include) is a decision which depends on the intended use of the diagram, you should consider

- Who is the audience?
- What are they using the diagram for?

The hyphens (-) before fields or methods indicates that member is private. The plus (+) means public. You can denote protected with an asterisk (*).

Relationship: Aggregation

- When an object has another object (typically through a pointer or reference) But that object is not owned.

If an object of type `A` "has-a" object of type `B`, then typically

- `B` has an existance outside of `A`
- If `A` is copied then `B` is not (shallow copy)
- If `A` is destroyed then `B` is not

e.g. `Courses` have `Students` and also `Students` have `Courses`.
Aggregation (also called a "has-a" relationship) is modelled in UML with a non-filled in diamond as above.

Relationship: Composition (Owns-a)

Composition relationship is when one object is composed of another (typically through containing a field of that type, or on owning a pointer)
Typically, if `A` owns-a `B`, then

- `B` has no existence outside of `A`
- If `A` is copied, then `B` is copied.
- If `A` is destroyed, then `B` is destryoed as well.

e.g. `Node`s own `Node`s
Multiplicities and names of relationships are optional (line fields and methods) but can be helpful.

Relationship: Specialization (Is-a)

A `B` is-an `A` if

- Everything you can do to an `A`, you can do to a `B`

Liskov Substitutability Principle

A `B` is only an `A` if it is substitutable in all situations for an `A`. Typically implemented through inheritance. e.g. `Student, regularStudent, CoopStudent`
Note: Abstract class typically modelled with italics. Also, italicize pure virtual functions.

The Vector Class

```
#include <vector>

int main() {
    vector<int> v; // empty vector of ints
    vector<int> m{5, 4}; // vector containing 5 and 4
    vector<int> l(5, 4); // 4,4,4,4,4
}
```

Note: here there is a difference between curly braces and parentheses.

```
vector<string> t(3, "Hello");
// vector that contains 3 "Hello"
```

Curly braces here take a list of elements you want in your vector.
Some vector operations
We can add things to the end of a vector with `vector::emplace_back`.

```
vector<int> v;
for (int i = 0; i < 20; ++i) {
    v.emplace_back(i);
}
```

Vectors can be indexed like an array.

```
Vector<int> v{1, 2, 3, 4};
v[0] = 20; // v is now 20, 2, 3, 4
```

Vectors also have iterator.

```
for (auto &x : v) {
    ++x;
}
```

BEWARE

```
void copy_and_change(vector<int> & v, int x) {
    auto it = v.begin();
    for (it, it != v.end() && *it != x; ++it);
    // make x copies of that number in v
    // and change the original to 0.
    for (int y = 0; y < x; ++x) {
        v.emplace_back(x);
    }
    *it = 0;
}
```

Iterator invalidation.
`it` is an iterator of the vector, it stores a pointer to the returned array, as we were adding things to the vector. Its array gets full, it had to allocate a new array. So the pointer in our `it` was dangling.
This problem is called iterator invalidation, you need to be aware of it to use a vector, documentation for vector methods will you if that method may invalidate iterators. So, be aware of it when using vectors and iterators.

13 June 14th

Vector manages the array it's built on, but not its contents.

```
class Foo {
    int * p;
    public:
        Foo() : p {new int} {}
        ~Foo() {delete p;}
};

int main() {
    vector<Foo> vf;
    vf.emplace_back();
    vf.emplace_back();
}
```

`vf` goes out of the scope and the vector destructor deletes the array.
`Foo` object destructors run.
However, consider the following:

```
int main() {
    vector<int*> vp;
    vp.emplace_back(new int{1});
    vp.emplace_back(new int{2});
}
```

vp goes out of scope, vector dtor deletes the array. The ints are leaked. You must delete the pointers.

Observer Pattern

Publish-Subscribe model
One class - Publisher/Subject - generates data One or more - Subscriber/Observer - receive data and react to it.

Example

Publisher: spreadsheet cells.
Observers: graphs
When the cells change, the graphs update themselves.
Can be many kinds of observer objects (e.g. formula cells)
The subject should not need to know all the details about them.
Sequence of method calls:

1. `Subject`'s state is updated.
2. `Subject::notifyObservers` is called (by `Subject` itself, or by an external controller)
3. `Subject::notifyObservers` calls `notify()` on each of subject's observers.
4. Each observer calls `ConcreteSubject::getState` to query the state, and then reacts accordingly.

Example

Horse races. The subject publishes winners. The observers are individual bettors. They will declare victory when their horse wins.

```
class Subject {
    vector<Observer *> observers;
    public:
        void attach(Observer *o) {
            observers.emplace_back(o);
        }
        void detach(OBserver *o) {
            // remove from vector.
        }
        void notifyObservers() {
            for (auto ob: observers) ob->notify();
        }
        virtual ~Subject() = 0; // Purely to make the class abstract
};

Subject::~Subject() {}

class Observer {
    public:
        virtual void notify() = 0;
        virtual ~Observer() {}
};

class HorseRace: public Subject {
    ifstream in;
    string lastWinner;
    public:
        HorseRace(const string &source):
            in {source} {}
        bool runRace() {
            return in >> lastWinner;
        }
        string getState() const {return lastWinner;}
};

class Bettor: public Observer {
    HorseRace * subject;
    string name, myHorse;
    public:
        Bettor(); ... {
            subject->attach(this);
        }
        ~Bettor() {
            subject->detach(this);
        }
        void notify() override {
            string winner = subject->getState();
            cout << (winner == myHorse ? "Win!" : "Lose. :-(") << endl;
```

```
        }
};

int main() {
    HorseRace hr{"source.txt"};
    Bettor Larry{&hr, "Larry", "RunsLikeACow"};
    // Other bettors
    while(hr.runRace()) {
        hr.notifyObservers();
    }
}
```

Let's see another example.

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

Exceptions

What if new fails? (Why might new fail? Can't satisfy the request for memory)
Exception is raised.

14 June 16th

`new` can detect the problem, but doesn't know what to do about it. Client knows what to do, but can't detect the error. Hence, error handling is an inherently non-local problem.

C solution

Functions return a status code, or set a global variable. It leads to awkward programming, encourages programmers to ignore errors.

C++ solution

When an error condition arises, the function raises an exception. By default, execution stops.
Or we can write handlers to catch exceptions and deal with the problem.
When `new` fails, it throws an exception of type `std::bad_alloc`

Example

Vectors - out of bounds index
`v[i]` - Undefined behaviour `v.at(i)` - raises a `std::out_of_range` exception.
Handling the exception:

```
#include <stdexcept>
...
// Statements that may raise an exception go into a "try" block
try {
    cout << v.at(10000) << endl;
```

```
} catch (out_of_range r) {
    cerr << "Range error: " << r.what() << endl;
}
// then go here
```

Now consider:

```
void f() {
    throw out_of_range {"f"};
}

void g() {
    f();
}

void h() {
    g();
}

int main() {
    try { h(); } catch (out_of_range ) { ... }
}
```

`main` calls `h`, `h` calls `g`, `g` calls `f`, `f` throws.
Control goes back through the call chain (unwinding the stack) until a handler is found. In this case, all the way back to main, main handles the exception.
If no handler - program terminates.
A handler can do part of the recovery job, i.e. execute some corrective code, throw another exception.

```
try { ... }
catch (SomeErrorType s) {
    throw SomeOtherError {...};
}
```

Or rethrow the same exception.

```
try { ... }
catch (SomeErrorType s) {
    ...
    throw;
}
```

Why `throw;` and not `throw s;`?
The type of `s` might actually be `SpecialErrorType`. `throw s;` throws a `SomeErrorType` (slicing)
`throw;` throw the exception that was caught and retain its exact type.
A handler can act as a `catch` all:

```
try { ... }
catch (...) {
    ...
}
```

You can throw anything you want, e.g. `int`s.
But generally - define your own classes (or use appropriate existing ones) for errors.

```
class BadInput {};

try {
    int n;
    if (!(cin >> n)) throw BadInput{};
}
catch (BadInput &) {
    cerr << "Input not well-formed\n";
}
```

We should throw by value and catch by reference. (This saves copying)
Also:

```
class BaseExn{};
class DerivedExn : public BaseExn {};

void f() {
    DerivedExn d;
    throw d;
}

int main() {
    try { f(); }
    catch (BaseExn &) { cout << "Base"; }
    catch (DerivedExn &) { cout << "Derived"; }
}
```

This prints `Base`. The first matching handler is executed.
Now consider the following:

```
void h() {
    DerivedExn d;
    BaseExn &b = d;
    throw b;
}

int main() {
    try { f(); }
    catch (DerivedExn &) {
        cout << "Derived";
    } catch (BaseExn &) {
        cout << "Base";
    }
}
```

Prints `Base` - exception handler chosen based on the static type of the data thrown. i.e., `BaseExn` (type of ref).

WARNING

Never let a destructor throw an exception! If an exception is raised, dtors run during stack unwinding. If one of these `throw`s, now have 2 active exceptions looking for a handler. Program will abort immediately.

15 June 21st

We care about the interface - and yet we can see these fields. Moreover, if these fields change (e.g. `rects` becomes a vector<SDL_Rect *>s) then all client code needs to recompile (recall compilation dependencies)
We don't care about these details, and don't want client to recompile anytime we want to change these internals.
Solution - pointer to implementation idiom (pImpl) idiom. (Create a new class "screenImpl" that stores the internal fields and just have `Screen` store a `ScreenImpl` pointer).

```
// ScreenImpl.h
struct ScreenImpl {
    SDL_Surface * screen;
    int w, h;
    int numRects;
    SDL_Rect ** rects;
}

// Screen.h
class Screen {
    ScreenImpl * pImpl;
    public:
        ...
}

// Screen.cc
Screen::Screen(...) : pImpl{new ScreenImpl (...)} {}

void draw_rect (int x, int y, int w, int h, Colour c) {
    if (pImpl -> numRects == 10) return;
    rects[numRects++] = SDL_Rect(...);
    SDL_Fill Rect(Screen, ...);
}
```

Only change we make is initializing object other methods now refer to fields as pImpl -> Field N are (or could make methods to impl class and just call them)
Back to exceptions.

```
int ** Foo(int x) {
    int ** arr = new int *[x]
    for (int i = 0; i < x; ++i) {
        arr[i] = new int{i};
        arr[i] = g(arr[i]);
    }
    return arr;
}

int main() {
    int ** x = foo(100);
    for (int i = 0; i < 100; ++i) delete x[i];
    delete[] arr;
}
```

Program is fine, no leaks - except if the inner new or if g throws, we've leaked everything already allocated. Ever if we recover from the exception foo propogates, we have a leak.
So, we can catch the exception in Foo and clean up memory.

```
int ** Foo(int x) {
    int ** arr = new int *[x];
    for (int i = 0; i < x; ++i) {
        try {
            arr[i] = new int{i};
            arr[i] = g(arr[i]);
        } catch (...) {
            for (int j = 0; j < i; ++i) {
                delete arr[i];
            }
            delete [] arr;
        }
        throw;

    }
}
```

Can still leak! If the ith new doesn't fail, but g does. Then that ith int is leaked. Could fix by writing two try catches, one for the new (that delects up to i) and one for the call to g that deletes up to and including i. But, that's very ugly, prone to errprs (Easy to miss cases) a lot of extra code to write. Some languages have a "finally' clause.
Code that is guaranteed to run no matter how a function exists. C++ has no such a feature: however, C++ promise the stack will be cleaned UP, and so the destructors of stack allocatied objects will run.
Resource Acquisition is Initialization (RAII) is a C++ idiom that states resources should only ever be acquired through the initialization of a stack based object whose job it is to manage it.
So, we can rewrite Foo...

```
class IntPtr {
    int **;
    public:
        IntPtr(int i): x{new int{i}} : {}
        int & operator*() return x * i;
        ~IntPtr() {delete x;}
        IntPtr(const Int ptr & other) : x{new int{*other}} {}
}
```

Now, let's rewrite foo

```
vector<IntPtr> foo(int x) {
    vector<Intptr> v;
    for (int i = 0, i < x; ++i) {
        v.emplace_back(IntPtr {i});
        v[i] = g(*v[i]);
    }
    return v;
}
```

Now, if `new` fails (int intptr's constructor) or if `g` fails, stack is unwound, the vector v is popped off, its destructor deletes its array, it's an array of IntPtr objects. So their dtor runs and frees the ints they point at, no leaks!
It's not practical to write an individual class for each type of pointer we want to store, so C++ provides to us classes for just that. e.g,. `std::unique_ptr` in the `memory` header.

```
vector<unique_ptr<int>> foo(int x) {
    vector<unique_ptr<int>> v;
    for (int i = 0; i < x; ++i) {
        v.emplace_back(unique_ptr<int> {new int {i}});
        v[i] = g(*v[i]);
    }
    return v;
}
```

`unique_ptr` is a STL class for managing pointers to data, but - as its name states it is unique - it believes it is the sole holder of that (or at least it maintains ownership) so warning.

```
int main() {
    int *p = new int{10};
    if (.....) {
        unique_ptr<int> {p};
        ......
    } // x goes out of scope, deletes p
    *p = 50; // memory error! *p is freed already.
}
```

This never would have happened if we followed RAII, also, ideally, we don't say `new` at all, instead we can use the function `make_unique` to allocate the data for us.
e.g. `unique_ptr<int> p = make_unique<int>(10)` This is a pointer to number 10
`make_unique`, like `emplace_back` takes arguments that match any ctor for the type it's storing (or the value itself for PODs)

```
void foo(unique_ptr<int> p) {
    ......
}
......

unique_ptr<int> q = make_unique<int> (10);
foo(q); // compiler complains, unique_ptr has no copy constructor or CAO

void foo(int *p) {
    ...
}

// q as before
foo(q.get());
```

`unique_ptr` get returns the underlying raw pointer.
Be careful not to store/use it for longer than the `unique_ptr` exists.
RAII helps us to write exception-safe code. There are 3 levels of exception safety that a function can offer.

1. The basic guarantee - if an exception is thrown or propogated by this function, then the program will be left in a valid but unspecified state. (i.e., no resources leaked, no class invariants violated, but the program state may have changed)

2. The strong guarantee - if the function throws, or propogates an exception, the state of the program will be as if it is never called. (nothing printed, values unchanged)

3. The no-throw guarantee - this function will always succeed, never throwing or propogating exceptions.

16 June 23rd

`vector::emplace_back` offers the strong guarantee. But `emplace_back` may have to allocate a new larger array. That means taking things from the old array and placing them in the new one.
Should it move construct the elements in the new array from the old? Ideally yes - it's faster. But if your move constructor can throw, the old array has been changed by the moves and `emplace_back` can no longer offer the strong guarantee.
So `vector::emplace_back` will use your copy constructor UNLESS your move constructor offers the no throw guarantee and tells the compiler as much by marking that method `noexcept`

```
class List {
    ...
    public:
        List(List && other) noexcept : head{other.head} {
            other.head = nullptr;
            // copying pointers never throws
        }
}
```

Ideally, all your move operations are `noexcept` - but how can we guarantee this?

```
class Foo {
    T a;
    Q b;
    P c;
    ...
}
```

`Foo` contains an `T`, `Q`, and `P`. I may not be able to swap all of these without the possibility of an exception. How can I guarantee that every class I write that I use can have non-throwing move?
Observation - swapping pointers never throws if we follow the pImpl idiom. All our objects only ever need to swap/assign their pImpl pointers and are guaranteed to not throw! A non-throwing swap is essential to writing exception safe code.

```
class Foo {
    T a;
    Q b;
    public:
        void set(T x, Q y) {
            a = x;
            b = y;
        }
}
```

At best, this is basic guarantee, but if changing my `a` without changing my `b` violates a class invariant then no guarantee (not even in a valid state).
BUT, if we have a non-throwing swap function following pImpl then we can use the copy-and-swap idiom to guarantee nothing changes or we succeed.

```
class Foo {
    FooImpl * pImpl;
    void swap(Foo & o) noexcept {
        std::swap(pImpl, o.pImpl);
    }
```

```
    public:
        void set(T x, Q y) {
            Foo tmp{*this};
            tmp.pImpl->a = x;
            tmp.pImpl->b = y;
            swap(tmp);
        }
}
```

Now (assuming `T` and `Q`'s `operator=` offers the strong guarantee), we do as well, however if `T` and `Q`'s `operator=` offers the basic or no guarantee, we can't undo what they've changed.

Decorator Pattern

The decorator pattern is used when we want to enhance/modify the behaviour of objects at runtime. So, for example, we have a Plain Screen, just a window. At runtime, we want the options to add the close/minimize button bar, also a scroll bar.

Example: Piazza

A pizza base is `crustAndSauce`, it can be decorated with `toppings` and `stuffed` crusts, that of these affects the price.

```
class Pizza {
    public:
        virtual float price() = 0;
        virtual string desc = 0;
        virtual ~Piazza() {};
};

class CrustAndSauce : public Pizza {
    public:
        float price() override { return 7.99; }
        string desc() override { return "Pizza"; }
};

class Decorator : public Pizza {
    Pizza * p;
    public:
        float price() override {
            return p->price();
        }
        string desc() override { return p->desc(); }
        Decorator(Pizza * p) : p {p} {}
        ~Decorator() { delete p; }
}

class Topping : public Decorator {
    string top;
    public:
        Topping (string t, Pizza * p) : Decorator{p}, top{t} {}
        float price() override {
            return 1.00 + Decorator::price();
```

```
        }

        string desc() override {
            return Decorator::desc() + " with " + top;
        }
}
```

17 June 28th

Templates

```
class List {
    struct Node;
    Node * next;
    public:
        ...
}
```

Node stored ints. What if we want a list that stores other things? Create a new type?
We can declare list as a template type. That is, a type that is parametrized with a type. (e.g. vector<int> v;) Here, we are parametrizing type vector with type int.

```
template <typename T>
class List {
    struct Node {
        T data;
        Node * next;
    }
    Node * head;

    public:
        class Iterator {
            Node * p;
            explicit Iterator (Node * p): p{p} {}
            public:
                Iterator & operator++() {
                    p = p->next;
                    return *this;
                }
                bool operator!=(const Iterator &o) {
                    return p != o.p;
                }
                T & operator*() {return p->data;}
        };
        void cons(T x) {
            head = new Node{x, head};
        }
};

// Client Code
List<int> l1;
l1.cons(3);
l1.cons(4);
```

```
List<char> l2;
l2.cons('a');
l2.cons('&');
List<List<int>> l3;
l3.cons(l1);
for (auto &l: l3) {
    for (auto &n: l) {
        cout << n << " " <<;
    }
    cout << endl;
}
```

Templated classes must have their method implementations in the header file.
What's happening (roughly speaking)? A templated class really defines a family of types. The compiler then generates specific forms of the class for each way the template class is specialized.
Also, what types are valid subsitutes for `T`? Consider:

```
template <typename T>
class Foo {
    T x;
    public:
        Foo operator+(const Foo & o) {
            return Foo{x + o.x};
        }
        T getX() { return x; }
};
```

What type `T` can `Foo` be parametrized with?

- `operator+` must be defined between `T`s.
- `T` must be copy initializable.

We cannot do: `Foo <unique_ptr<int>> f;` `unique_ptr`'s are not copyable (or addable for that matter).

```
template <typename T>
class unique_ptr {
    T * p;
    public:
        unique_ptr(T x) : p{new T{x}} {}
        ~unique_ptr() {delete p;}
        unique_ptr(const unique_ptr<T> & o) = delete;
        unique_ptr & operator=(const unique_ptr<T> &) = delete;
}
```

The valid template specializations are whatever matches the way your template class uses that type! This is called duck typing.
If it walks like a duck and quacks like a duck - then it is a duck.

18 June 30th

Template Functions

```
template <typename Iter, typename Ret>
```

```
Ret sum(Iter start, Iter end) {
    Ret sum{};
    while (start != end) {
        sum += *start;
        ++start;
    }
    return Ret;
}

int main() {
    vector<int> x{1, 2, 3, 4};
    cout << sum<vector<int>::iterator, int>(x.begin(), x.end());

    int arr[5]{1, 2, 3, 4, 5};
    cout << sum<int *, int>(arr, arr+5);
}
```

Template classes must always parametrize the instantiations - template functions you do not have to do this IF the compiler can infer the types.

```
template <typename Iter, typename Func>

void for_each(Iter start, Iter end, Func f) {
    while(start != end) {
        f(*start);
        ++start;
    }
}

void print(string s) {cout << s << endl;}

int main() {
    vector<string> v{"Hello", "There"};
    for_each(v.begin(), v.end(); print);

    // don't need to parametrize template
    // here, compiler infers param types from
    // static types of args
}
```

`for_each` and functions like it are defined in the STL in the `<algorithm>` header (most of them operate on iterators). Familiarize yourself with this header, it has many useful functions.
What types `T` can be substituted for `Func`? - Any types which are callable as a function (also the param type must match type of dereferencing iter.)
What other things than functions are callable as functions?
Anything which defines `operator()`.
I want a function that adds 7 to its int parameter. I also want a function that adds -10 to its int parameter. And I may want more of these specific add functions at any time.
Should I write a separate function for each of these?

```
class Adder {
    int x;
    public:
```

```
        Adder(int x) : x{x} {}
        int operator()(int x) {
            return x + n;
        }
};

int main() {
    Adder add10{10};
    Adder sub5{-5};
    cout << add10(7) << sub5(10) << endl;
    // 17 5

    for_each(v.begin(), v.end(), Add{15});
}
```

I want a function that adds 1 to its param the first time it's called, the second time it's called I want it to add 2, third time 3, etc...

```
class IncreasingPlus {
    int x = 1;
    public:
        void operator()(int & n) {
            n+= x++;
        }
        IncreasingPlus a;
        vector<int>{0, 0, 0, 0};
        for_each(v.begin(), v.end(), a);
        // v is now {1, 2, 3, 4};
        for_each(v.begin(), v.end(), a);
        // v is now {6, 8, 10, 12};
}
```

Objects that are callable as functions are called function objects (some people call them functors, but that name has other connotations in other uses so I don't like it)
Classes like this allow us to define families of functions (e.g. `Adder`) or/and functions with state.
Also, we have another option for short functions we don't want to write lambda functions! - a lambda function is an anonymous function (doesn't have a name)

Lambdas in C++

e.g.

```
vector<int> v{1, 2, 3, 4};
for_each(v.begin(), v.end(), []->int(int m) {return n+7;})
```

What if we have an `int x` in main. We want to add to each element? e.g.

```
int x;
cin >> x;
for_each(v.begin(), v.end(),
    [&x]->int(int n){ return n + x;});
```

The compiler makes an object for each lambda you create.

19 July 5th

Entity vs. value ADTs

A Entity ADT represents a real world entity. A value ADT represents some kind of abstract values.
Examples of entity ADTs - Books, Students, Piazzas, etc.
Examples of value ADTs - lists, forces, points, Rational numbers expressions, etc.
Typically, Entity ADTs

- Disallow copying, or think twice before you write a copy (what should the behaviour be)
- Think before you overload operators, most don't make sense
- Methods on entities typically represent real world events
- Two entities are NOT equal if they have same values, typically only equal if they are the same object (i.e., same location in memory)

Value ADTs typically

- allow copying
- are equal if they have the same values
- often have many overloaded operators

These decisions all come down to design and what makes sense for your system.
License Plate ADT could make a valid argument either way. We'll create a value ADT for license plate.
Our simplified requirements
A license plate will be

- 3 numbers followed by 3 letters
- vanity plate, string of up to 7 characters, minimum of 3.
- Default ctor that builds next license plate.
- Parametrized ctor with string that constructs vanity plate. Disallow hyphens in vanity plates.

```
class License {
    // What is a static field?
    static string digits;
    static string letters;
    string license;
    static void nextLicense();
    // static methods do not reply
    // on an specific instance of an object,
    // that is they do not have an implicit
    // this parameter, they can only refer
    // to other static variables/methods
    public:
        License();
        License(string s);
}
```

A static field is a field that belongs to the class, not an individual object. It's shared across all objects of that type.
Where to initialize?

- Can't initialize it in the class, C++ standard says so (*)
- Definition and initialization must go in the `.cc` (implementation) file.

```
// license.cc

#include <sstream>
#include <iomanip>

string License::digits = "000";
string License::letters = "aaa";

int intChar(char &c) {
    // pre 'a' <= c <= 'z'
    ++c;
    if (c > 'z') {
        c = 'a';
        return 1;
    }
    return 0;
}

void License::nextLicense() {
    istringstream iss{digits};
    int num;
    iss >> num;
    ++num;
    if (num == 1000) {
        num = 0;
        if (intChar(letters[2])) {
            if (intChar(letter[1])) {
                intChar(letters[0])
            }
        }
    }
    ostringstream oss;
    oss << setfill('0') << setw(3) << num;
    letters = oss.str();
}

License::License() : license{letter + "-" + digits} {
    nextLicense();
}

License::License(string s) : license{s} {
    if (s.length < 3 || s.length > 7) throw BadLicense{};
    auto it = find(s.begin(), s.end(), '-');
    if (it != s.end()) throw BadLicense{};
}
```

Shared Pointers

We've talked about `unique_ptr`, and using `unique_ptr<T>::get` to set up non-owning pointers to the same data. But - what about the case where we have shared ownership?

In the <memory> header, we also have std::shared_ptr<T>
Usage:

```
shared_ptr<int> sp = make_shared<int>(5);
if (...) {
    shared_ptr<int> t{sp};
    ...
} // t goes out of scope, does delete the
  // int it points at? No, sp still points at it

// Now sp goes out of scope, does free that memory

template <typename T>
class sharedptr {
    T * p;
    size_t *count;
    public:
        sharedptr(T *p) : p{p}, count{new size_t{1}} {}
        sharedptr(const sharedptr & p) : p{p.p},
        count{p.count} {
            ++*count;
        }
        ~sharedptr() {
            --*count;
            if (!*count) {
                delete p;
                delete count;
            }
        }
        sharedptr<T> & operator=(const sharedptr & p) {
            if (&o == this) return *this;
            --*count;
            if (!*count) {
                delete count;
                delete p;
            }
            p = o.p;
            count = o.count;
            ++*count;
            return *this;
        }
}
int main() {
    sharedptr<int> sp1{new int{10}};
    if (true) {
        sharedptr<int> sp2{sp1};
    }
}
```

20 July 7th

Template Method Pattern

Problem: We want a function for which some behaviour can be specialized, but other behaviour remains the same.
e.g. We want to draw Red and Green turtles, but the head and feet should always be drawn the same

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

class RedTurtle : public Turtle {
    void drawShell() {
        // draws a red Shell
    };
};

class GreenTurtle : public Turtle {
    void drawShell() override {
        // draws a green shell
    }
};
```

So now, all turtle's feet and head are drawn the same, only the shell part can be specialized.
This is the template method pattern.

Non-virtual interface idiom

The non-virtual interface (NVI) idiom suggests that all virtual methods should be wrapped up in non-virtual public wrappers that call them. Why?
A public virtual method is really two things:

- An interface to the client (public)
  - Promises some kind of behaviour, i.e., post conditions, class invariants, etc.
- An interface to the derived classes (virtual)
  - A hook for the derived classes to inject specialized behaviours

How can the Base class make a promise to the client when it doesn't control what code executes? e.g. `DigitalMedia`

```
class DigitalMedia {
    public:
        virtual void play() = 0;

};
```

Derived classes override that to play their various media types.
Disney sues you, your software is not checking DRM. Now, all play methods must check DRM before playing.
Rather than adding the check to each derived class's play methods, if we follow NVI, we can ensure `checkDRM` is called in our non-virtual play methods before dispatching to a virtual helper.

```
// Translated to NVI
class DigitalMedia {
    virtual void doPlay() = 0;
    void checkDRM() { ... }
    public:
        void play() {
            CheckDRM();
            doPlay();
        }
}
```

This guarantees DRM is always checked first regardless of what derived classes do.
Saves us lots of duplicated code and possible mistakes. Regaining this control after the fact (if code has already been written) is much harder and tiem consuming than using NVI from the beginning.
The NVI idiom states:

- All public methods should be non-virtual
- All virtual methods should be private or protected
- Except, of course, the destructor.

Another STL Class: Map - for creating dictionary

```
#include <map>
using namespace std;
map<string, int> phoneBook;
phoneBook["Abdur"] = 7224687;
cout << phoneBook["Abdur"] << endl; // 7224687
cout << phoneBook["Rob"] << endl; // 0
```

If key doesn't exist, creates and default initializes the value.
NOTE: Yes, integral types have a default initializer. But

```
int n; // Not default initialized, uninitialized data still
int x{}; // x is default initialized to 0
```

If indexing the map adds the key, how to check if a key exists?

```
if (phoneBook.count("Brad"))
// count is 0 if doesn't exist, 1 if it does

auto it = find(phoneBook.begin(),phoneBook.end(), "Brad");
if (it != phoneBook.end()) // key is found
// iterating over a map.
for (auto &p : phoneBook) {
    cout << p.first << " " << p.second << endl;
}
```

Type of `p` is `std::pair<string, int> &`, `std::pair` is declared in `<utility>`, you access the fields first and second.
Iterating over a map iterates in sorted key order.
If you don't care about the order in which your map is iterated, use `unordered_map<T, P>` from header `<unordered_map>`, most of its methods are quicker.

21 July 12th

Double Dispatch

We have two hierarchies, and I have a function whose behaviour depends on BOTH objects.
e.g. Striking enemies with weapons
Striking an enemy with a weapon depends both on the weapon type and the enemy type.

```
Enemy * e = ....;
Weapon * w = ....;
w->strike(*e); // Chooses based on weapon not enemy
e->struck(*w); // vice versa
```

So, if I have `virtual void Enemy::Struck(weapon &w)`, then I choose only based on Enemy type if I have `virtual void weapon::strike(Enemy &e)`, then vice versa.
What I want is something like `virtual void (Enemy, weapon).strike()` but no such thing exists. One solution to this problem is the visitor pattern - and the key to this is combining overrides and overloads.

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

class Stick : public Weapon {
    public:
        void strike(Enemy & e) {
            e.beStruckBy(*this);
        } // Strike an enemy with a STICK
};

class Rock : public Weapon {
    public:
        void strike(Enemy & e) override {
            e.beStruckBy(*this);
        } // Strike an enemy with a ROCK
};

class Enemy {
    public:
        virtual void beStruckBy(Rock &) = 0;
        virtual void beStruckBy(Stick &) = 0;
};

class Turtle : public Enemy {
    public:
        void beStruckBy(Stick &s) override {
            // Code to strike a turtle with a stick
```

```
        }
        void beStruckBy(Rock &r) override {
            // Code to strike a turtle with a stick
        }
};

class Bullet {
    // Similar
};

Enemy * e = new Bullet{...};
Weapon * w = new Stick{...};
w->strike(*e);
```

- virtual dispatch calls `Stick::strike`
- `Stick::strike` calls `e.beStruckBy(*this)`. `this` is a `Stick` pointer, so virtual dispatch is done on `beStruckby(Stick &)` method.
- virtual dispatch chooses a `Bullet::beStruckBy(Stick &)` and that tuns the code for striking a bullet with a stick.

The visitor pattern can be used to add additional functionality to a hierarchy without changing it itself.

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

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

class Text : public Book {
    // similar
};

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

struck Catalogue : public BookVisitor {
    map<string, int> theCatalogue;
```

```
    void visit(Book &b) {
        ++theCatalogue[b.getAuthour()];
    }
    void visit(Text &t) {
        ++theCatalogue[t.getTopic()];
    }
    void visit(Comic &c) {
        ++theCatalogue[c.getHero()];
    }
}
```

Measure of Design Quality

Coupling - how tightly intermingled your modules are, how much one class relies on the details of another.

High Coupling	classes access and use private data within each other.
	Classes communicate through mutating global variables
	classes communicate back and forth with specific specialized types.
Low Coupling	classes interact through the public interface of each other, with functions that consume and return basic data type

Goal is low coupling
Cohesion is the degree to which the content of a module "belong together".

Low Cohesion	Collection of random class and methods (e.g. Everything in one file)
	Collection of assorted functions and classes with a common theme, and maybe some common code (e.g. <algorithm>)
	classes work together in order to provide various behaviours (example: module with data class and dispaly class).
High Cohesion	contents of module (classes/functions) work together to achieve a signle task (e.g. <vector>)

Desire high cohesion

Casting in C++

First of all, casting is usually the wrong thing to do, usually indicates a design flaw, but if you must cast, use C++-style casts, not C casts. They are safer.
`static_cast` for meaningful cast with well defined conversions. e.g., int to float.

```
int x = 5;
float f = static_cast<float>(x);
```

`const_cast` is for casting away "constness". The only real use is for calling a function which didn't declare it's parameter const, but doesn't actually change it.

```
void Foo(A & a) {
    // doesn't actually mutate a
}

const A a {...};
Foo (const_cast<A &> (a));
```

`reinterpret_cast` - for not well defined "weird cast" - almost all uses of `reinterpret_cast` lead to undefined behaviour - basically you're telling the compiler "trust me".

22 July 14th

`dynamic_cast` is for casting from base class pointers or references to derived class pointers or references - effectively allows you to find out what a base class pointer actually points at.

```
void f(AbstractBook * p) {
    if (dynamic_cast<Comic *>(p)) {
        cout << "Comic" << endl;
    } else if (dynamic_cast<RegularBook *>(p)) {
        cout << "Regular Book" << endl;
    } // same for Text
}
```

`dynamic_cast` on a pointer returns the `nullptr` if that pointer doesn't actually point at that object, if it does, it returns back the pointer itself.
Code like `f` is highly coupled to the `Book` hierarchy, also it defeats the purpose of inheritance and polymorphism altogether. We're meant to not care what these things point at, and work through the abstract interface.
So, in general, most uses of `dynamic_cast` indicate a flawed design, and are a bad code style.
Back to the polymorphic assignment problem... If for some reason, you really want/need to allow polymorphic assignment, it is now possible with `dynamic_cast`. E.g.

```
class AbstractBook {
    ...
    public:
        virtual AbstractBook & operator=(const AbstractBook & b) {
            authour = b.authour;
            title = b.title;
            numPages = b.numPages
            return *this;
        }
};

class Comic : public AbstractBook {
    public:
        Comic & operator=(const AbstractBook & o) override {
            Comic & c = dynamic_cast<const Comic &>(o);
            // Throws an exception if o doesn't actually refer to a Comic object
            AbstractBook::operator=(o);
            hero = c.hero;
            return *this;
        }
}
```

Still, the recommendation hasn't changed, 99% of the time assignment through base class pointer or references doesn't make sense, but if you must do it, that is how.
All of these costs, have smart poitner variants, so if you must do this on smart pointers, use:
`dynamic_pointer_cast` and `static_pointer_cast`.

Multiple Inheritance

Our Abstract Base Classes represent common interfaces that type share (Also encapsulate common data and common code). Because of this, it's not abnormal to have classes that inherit from more than one Abstract Base Classes.
For example:

```
class Drawable { // something that can be drawn on the screen
    public:
        virtual void draw(int x, int y) = 0;
}

class Controllable { // things controllable through input
    ...
    public:
        virtual void handleInput(InputEvent & e) = 0;
}
```

Our player character is both drawable AND controllable.

```
class PlayerCharacter: public Drawable, public Controllable {
    ...
    public:
        void draw(int x, int y) override { ... }
        void handleEvent(InputEvent & e) override { ... }
}
```

This is called multiple inheritance, and it's fine and allowed by C++. BUT there are problems. So, e.g. The diamond of death, the deadly diamond problem, the deadly diamond of death.

```
D dObj;
dObj.x = 10;
```

This is an error, if it wasn't an error, it would be ambiguous. `D` has two `x` fields, one from `B` and one from `C`.
So, I must say

```
D dobj;
dobj.B::x = 10;
dobj.C::x = 10;
```

Is this what we really want? Do we really want to inherit two `x` fields? Probably Not.
The solution to this Problem in C++ is virtual inheritance.

```
class A {
    public:
        int x;
};

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

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

class D : public virtual B, public virtual C {
    ...
};

D dObj;
dObj.x = 10;
```

Now, `D` has only one `x` field, the one inherited from its ancestor `A`.

23 July 19th

Singleton Design Pattern

Singleton design pattern is an antipattern, typically a bad idea to use it, indicates a flawed design.
The "problem" is when you only want one of an object in your entire program.
Often, when that class manages a piece of hardware, you know you're going to have one of.
What it's actually used for is "I want a bunch of global data but want to look like I'm not doing that".
Bad example, (don't do this) you're writing a game and you think you should only ever have one of these settings objects, so you use the singleton pattern.

```
class Settings {
    class SettingsImpl {
        int w, h;
        ....
    }
    static unique_ptr<SettingsImpl> p;
    public:
        Settings() {
            if (p.get() = nullptr) {
                p = make_unique<SettingsImpl>(...);
            }
        }
};

unique_ptr<SettingsImpl> Settings::p = nullptr;

class Character {
    ...
    public:
        void draw() {
            Settings s;
            int w = s.getw();
            int h = s.geth();
        }
};

int main() {
    Settings s;
    // first time, so settings p is setup
}
```

Alternatively, ...

```
class Singleton {
    int w, h;
    ...
    static Singleton * instance;
    // All ctors are private
    public:
        static Singleton *getInstance() {
            if (!instance) {
                instance = new Singleton(...);
            }
```

```
            return instance;
        }
}

int main() {
    Singleton * p = Singleton::getInstance();

}
```

Curiously Recurring Template Pattern (CRTP)

Problem: You want dynamic binding (i.e. virtual dispatch) without all that pesky virtual overhead - you want "virtual" dispatch at compile time.
Pros - shared common interfaces with different implementations of those interfaces.
Cons - no polymorphism (e.g. Collection of abstract base class pointers)

```
template <typename Derived>
class Base {
    public:
        void dOSomething() {
            // interface
            static_cast<Derived *>(this)->doSomethingImpl();
        }
};

class DerivedOne : public Base<DerivedOne> {
    public:
        void doSomethingImpl() {
            ...;
        }
};

class DerivedTwo : public Base<DerivedTwo> {
    ...
    public:
        void doSomethingImpl() {
            ...;
        }
};

class DerivedThree : public Base<DerivedOne> {
    ...
    public:
        void doSomethingImpl() {
            ...
        }
}

DerivedThree d;
doSomething(); // calls DerivedOne doSomethingImpl and
// but d is not a DerivedOne
// undefined behaviour
```

Observation: Derived classes must instantiate their base class component.
Solution: Make all constructors of base class private and declare the template param class as a friend!

```
template <typename Derived>
class Base {
    // All ctors here private
    public:
        void dOSomething() {
            // interface
            static_cast<Derived *>(this)->doSomethingImpl();
        }
        friend class Derived;
};
```

24 July 21st

Template Parameter Packs

For a variadic amount of arguments.

```
template <typename T, typename ... Args>

void print(T first, Args ... rest) {
    cout << first << endl;
    print(rest...);
}

template <typename T>
void print(T first) {
    cout << first << endl;
}
```

Namespace

```
// vector.h
namespace CS247 {
    template <typename T>
    class Vector {
        size_t size, capacity;
        T * arr;
        public:
            void emplace_back(const T & x);
    };

    template <typename T>
    void Vector<T>::emplace_back(const T & x) {
        if (size == cap) {
            doubleMem(*this);
        }
        arr[size++] = x;
    }
}
```