Final Review

CS 246 Fall 2019

December 2019

Contents

1 Encapsulation, System Modelling **3**
- 1.1 `friend` Keyword . . . 3
- 1.2 Setters and Getters . . . 3
- 1.3 I/O Operators and Friendships . . . 3
- 1.4 System Modelling . . . 4

2 Inheritance **5**
- 2.1 Steps for Object Creation (Updated) . . . 5
- 2.2 `protected` Keyword . . . 6
- 2.3 Method Overriding . . . 7
- 2.4 Object Slicing/Coercion . . . 7

3 Destructor, Virtual, Template **7**
- 3.1 Destructors with Inheritance . . . 7
- 3.2 Pure Virtual . . . 8
- 3.3 C++ Templates . . . 10
- 3.4 STL: Standard Template Library . . . 11

4 Design Patterns **12**
- 4.1 Revisiting the Interator Design Pattern . . . 12
  - 4.1.1 Example of coding to an interface . . . 13
  - 4.1.2 Example . . . 13
- 4.2 Observer Pattern . . . 14
  - 4.2.1 Example - SpreadSheets . . . 14
- 4.3 Decorator Pattern . . . 15
  - 4.3.1 Example: Browser Window . . . 16
- 4.4 Factory Method Pattern . . . 16
  - 4.4.1 Virtual Constructor Pattern . . . 17

5 Exceptions **17**
5.1 Example . . . 17
5.2 Better Example . . . 17
5.3 Error Recovery . . . 18
5.3.1 Example . . . 18
5.4 Exception Classes . . . 19
5.4.1 Example . . . 19
5.5 Some types of Exceptions . . . 19

6 Big 5 Revisited **20**

7 Design Patterns (Again) **22**
7.1 Template Design Pattern . . . 22
7.2 NVI (non-virtual interface) Idiom . . . 23
7.3 STL: map template class . . . 23
7.4 Visitor Design Pattern . . . 24

8 Compilation Dependencies, Design Strategies **25**
8.1 Compilation Dependencies . . . 25
8.2 Design Strategy . . . 26
8.2.1 pImpl Idiom . . . 26

9 Exception Safety **27**
9.1 Exception Safety . . . 27
9.2 C++ Guarantee . . . 28
9.3 RAII: Resource Acquisition Is Initialization . . . 28
9.4 Three Levels of Exception Safety . . . 29
9.5 STL: Vector class . . . 30

10 Casting **30**
10.1 Static Cast . . . 31
10.2 Const Cast . . . 31
10.3 Dynamic Cast . . . 31
10.4 Reinterpret Cast . . . 32
10.5 Downcasting . . . 32
10.6 Run-Time Type Information (RTTI) . . . 32

1 Encapsulation, System Modelling

1.1 `friend` Keyword

Problem: We would like to make the iterator class private (so users cannot access it). However, the List class would also not be able to access it.
Solution: declare List as a friend

```
class List {
    // code
    public:
        class Iterator {
        //code
        Node *p;
        Iterator(Node *p);
        //code
        public:
            //code
            friend class List;
        };

};
```

- Have as few friends a s possible (since having friends breaks encapsulation)
- Keep all fields private

1.2 Setters and Getters

Implement acessors/getters and mutators/setteres to provide read/write access. If you have an invariant about private fields, you can enforce them using the getters/setters.

1.3 I/O Operators and Friendships

- `operator<<` and `operator>>` have to be stand-alone functions and need access to the fields

Problem with using getters/setters: while the I/O operators can use these fields, any other user can use them as well. Solution: Class can declare the I/O operators as friends

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

```
ostream &operator(ostream &, const Vec &);
```

1.4 System Modelling

- Identify the key abstractions/entities/classes
- Identify the relationship between classes
- **UML**: Unified Modelling Language

Relationship 1: Composition (OWNS-A)

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

class Basis {
    Vec v1, v2;
};

Basis b; // this won't compile since default constructing v1, v2 requires Vec's
    default constructor
```

Option 1: Provide a default constructor Option 2: Bypass the call to the default constructor by calling a different constructor in the MIL

```
class Basis {
    Vec v1, v2;
    public:
        Basis(): v1{0,0}, v2{1,1} {}
};
Basis b; // compiles
```

Basis OWNS-A Vec. A OWNS-A B if:

- whenever A is copied, B is copied (deep)
- whenever A is destroyed, B is destroyed

Relationship 2: Aggregation (HAS-A) We say A HAS-B if:

- when A is copied, B is not (shallow)
- when A is destroyed, B is not

Relationship 3: Inheritance (IS-A)

2 Inheritance

How can we show the relationship of inheritance in C++ code?

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

class Text : public Book { // syntax to say Text IS-A Book, same as extends in
    Java
    string topic;
    // code
};
```

Because `Text` IS-A `Book` object, we can do anything with a `Text` object that we could with a `Book` object. For example, `ifstream` IS-A `istream`.

Subclasses inherit ALL members from the superclass.

```
int main {
    Text t{};
    t.author = "";
    // above line will not compile, since author is private from the Book class
}
```

Let's (attempt to) write the constructor for `Text`.

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

The above constructor will compile.

1. title, author, numPages are private in Book
2. MIL can only refer to fields declared in that class
3. Unable to default construct the superclass part of the object (however it would not make sense to set default values for a Book)

2.1 Steps for Object Creation (Updated)

1. Space is allocated

2. Superclass part is constructed
3. Subclass fields are constructed
4. Constructor Body runs

The constructor below will fix all three issues from before, by using the superclass constructor for superclass fields.

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

2.2 `protected` Keyword

- Accessible to the class and its subclasses

```
class Book {
    protected:
    string author;
    //////
};
class Text : public Book {
    //////
    void addAuthor(string a) { author += a}
};
int main() {
    Text t{};
    t.author = x;
}
```

Using `protected` breaks encapsulation.

- Subclasses can break invariants

```
class Book {
    string author;
    protected:
    void addAuthor(string a) {
        // invariant check on value "a"
        author += a;
    }
};
```

2.3 Method Overriding

You can write methods with the *exact* same signature, as long as they're in different classes.

- Book is heavy if numPages > 300.
- Text is heavy if numPages > 500.
- Comic is heavy if numPages > 30.

```
class Book {
    protected:
    int numPages;
    public:
    bool isHeavy() const;
};

bool Book::isHeavy() const { return numPages > 300; }
bool Text::isHeavy() const { return numPages > 500; }

////

Book b{"Title", "author", 200};
b.isHeavy(); // Book::isHeavy, false
Comic c{"Batman", "Nomair", 40, "Nomair"};
c.isHeavy(); // Comic::isHeavy, true
Book b = Comic{stuff} // this line calles the Book's copy constructor
b.isHeavy() // Book::isHeavy, false
```

2.4 Object Slicing/Coercion

```
Book *bp{&c};
bp->isHeavy();
```

No object slicing, this line calls Book::isHeavy because the compiler says so, and looks at the declared type of bp.

3 Destructor, Virtual, Template

3.1 Destructors with Inheritance

Steps for object destruction:

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

Examples:

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

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

// Run with Valgrind
int main() {
    X x{5};
    Y y{5, 10};

    X *xp = new Y{5, 10};
    delete xp;
}
```

Problem:Which destructor runs? X's destructor runs as xp points to x*. However, this deallocates int *x, and not int *y. y leaks!

Solution: Make destructor virtual. Tip: Always make the base class's destructor virtual. This applies even when the default constructor runs.

If a class will never have subclasses, declare the class "find".

```
Class Y find public X {
};
```

Note: `finds` is a keyword in this context only.

3.2 Pure Virtual

```
class Student {
    virtual int fees();
};

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

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

// Student::fees() has no implementation. The compiler will throw an error
    unless we use a pure virtual method.

class Student {
    virtual int fees() = 0; // Now this is a pure virtual function!
}
```

A subclass MUST implement pure virtual methods to be considered concrete.

```
Student s{};
```

This does not compile because `fees()` is not implemented.

- Class is incomplete
- This class is **abstract**

What makes a class abstract? A class is abstract if

- it declares a pure virtual method
- it inherits a pure virtual method that it does not override

If a class is not abstract, it is **concrete**.
Abstract base classes are useful for:

- organizing subclasses
- declaring fields/methods common to all students
- still using polymorphism
  i.e We can declare a `Student *student[3600];` We don't know what kind of student they are.

UML

- Virtual/PV methods - italics
- abstract - class name in italics
- protect - #

3.3 C++ Templates

```
// Stack of ints
class Stack {
    int *contents;
    int length;
    int capacity;
    public:
        Stack();
        void push(int x);
        void pop();
        int top();
        ~Stack();
};
```

But what if we want a stack of strings, polynomials, etc.? C++ Template Class is parameterized on one or more types.

```
template <typename T>

class Stack {
    T *contents;
    int length;
    int capacity;
    public:
        Stack();
        void push(T x);
        void pop();
        T top();
        ~Stack();
};

Stack s; // Does not compile. Expects a type but no type is provided.

Stack <int> s; // Defines a stack of integers. Intuitively, this copies code +
    replaces with int.
s.push(1);

Stack <char> s2; // This occurs at runtime
```

```
s2.push('o');
```

// Recall our list with Iterator

```
template <typename T>

class List {
    struct Node {
        T data;
        Node *next;
    };
    Node *theList = nullptr;

    public:
        class Iterator {
            Node *curr;
            Iterator(Node *)
            public:
                T operator*(); // Return type T
                Iterator &operator++();
                bool operator!=(...);
                friend class List<T>; // Whenever we reference List, we must do
                    List<T>
        };
        void addInFront(T *);
        T itr(int i);
        ~List();
};
List<int> li;
li.addInFront(1);

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

3.4 STL: Standard Template Library

STL is a collection of standard template classes. For dynamic length arrays, we can use `std::vector`

- parameterized as one type
- stack allocated

```
#include <vector>
vector<int> v{3,4}; // [3,4]
```

```
v.emplace_back(s) // [3,4,5]
```

Vectors internally use a heap array. How do we iterate over vectors?

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

We can also do:

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

for (auto &n: v) {
    ++n;
}

// Reverse iteration
for (vector<int>::reverse_iterator it = r.begin(); it != v.end(); ++it) {
    ...
}
v.pop_back(); // We can use vectors as a stack
v.erase(v.begin()); // Remove first element - like a queue
v.erase(v.begin() + 3); // Remove first element. Erase invalidates existing
    Iterators
```

4 Design Patterns

Now that we have discussed abstract classes, we can move further with design patterns. Many patterns share the same strategy: program to an interface, not an implementation

- Create abstract base classes to define the interface
- Work with pointers to these abstract base classes - recall that they can point to any concrete subclasses

4.1 Revisiting the Interator Design Pattern

Recall our TODO list: `operator*`, `operator++`, `operator!=`. We will provide an abstract Iterator that provides an interator

```
class AbsItr {
    public:
        virtual int &operator*() const = 0;
```

```
        virtual AbsItr operator++() const = 0;
        virtual bool operator!=(const AbsItr &) const = 0;
        virtual ~AbsItr() {}
};

class List {
    struct Node;
    Node *theList = nullptr;
    public:
        class Iterator : public AbsItr {
            // override the inherited methods
        };
};

class Set {
    // some code
    public:
        class Iterator : public AbsItr {
            // override the inherited methods
        };
};
```

4.1.1 Example of coding to an interface

```
template <typename Fn> // function or parameter
void foreach(AbsItr &start, const AbsItr &end, Fn f) {
    while (start != end) {
        int &tmp = *start;
        f(tmp);
        ++start;
    }
}
```

`foreach` uses the functions of what AbsItr points at! This is useful because

- The types of start and end are abstract (we can use `foreach` for both list and set objects)
- Uses `++`, `!=`, `++`

How do we call this?

4.1.2 Example

```
void addFive(int &n) { n += 5; }
////
List::Iterator lbegin = list.begin();
foreach(lbegin, list.end(), addFive); // the compiler recognizes the type of f,
    we pass addFive as a parameter
Set::Iterator sbegin = set.begin();
foreach(sbegin, set.end(), addFive);
```

Some observations:

- The function must be one parameter
- We could use templates for any type

4.2 Observer Pattern

- Push notification use observer pattern - tells you when an event occurred
- We call this a publish/subscribe model
  - The publisher is the subject
  - The subscriber is the observer

4.2.1 Example - SpreadSheets

- When we change a data point, the table automatically updates
- The table is the observer
- The cells are the subjects

The UML is as follows: Subject and Observer are the base classes, and Subject HAS-A Observer. ConcreteSubject and Concrete Object inherit from their respective base classes. ConcreteObserver HAS-A ConcreteSubject. ConreteSubject must have a public `getState()` method.

```
// subject.h
class Subject {
    vector<Observer *> observers;
    public:
        Subject();
        void attach(Observer *o);
        void detach(Observer *o);
        void notifyObservers();
        virtual ~Subject() = 0; // to make the class abstract
};
```

```
// subject.cc
Subject::Subject() {}
Subject::~Subject() {} // must be implemented even though it's abstract

void Subject::attach(Observer *o) { observers.emplace_back(o); }

void Subject::detach(Observer *o) {
    for (auto it = observers.begin(); it != observers.end(); ++it) {
        if (*it == o) {
            observers.erase(it);
            break;
        }
    }
}

// observer.h
class Observer {
    public:
        virtual void notify() = 0;
        virtual ~Observer();
};

// observer.cc
Observer::~Observer() {}

// horse.cc, merged with .h for conciseness
class HorseRace : public Subject {
    std::fstream in;
    std::string lastWinner;

    bool runRace() {
        bool result = in >> lastWinner;
        if (result) std::cout << lastWinner;
        return result;
    }

    std::string getState() { return lastWinner; }
};
```

4.3 Decorator Pattern

- Decorate objects - modify the behaviour of existing obejcts
- Example: Adding a toolbar to the browser

Heirarchy is as follows:

- BaseAbstractObject - public operation
  - BaseObject : public BaseAbstractObject - public operation
  - Decorator (Abstract) : public BaseAbstractObject - HAS-A BaseAbstractClass
    - Decoration1 : public operation
    - Decoration2: ...

4.3.1 Example: Browser Window

```
AbsWindow *p = new ConWindow; // base object
if (user adds scrollbar) {
    p = new Scrollbar{p}; // decoration
} else if (user adds toolbar) {
    p = new Toolbar{p}; // decoration
}

p->render(); // recursive, render is the operation
```

4.4 Factory Method Pattern

Consider the following example classes:

- Enemy (Abstract)
  - Turtle
  - Bullet
- Level (Abstract)
  - Normal
  - Castle

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

while (p->notDead()) {
    e = l->createEnemy();
}

class Level {
```

```
    virtual Enemy *createEnemy() = 0;
};

Enemy *Normal::createEnemy() override { more turtles }
Enemy *Castle::createEnemy() override { more bullets }
```

4.4.1 Virtual Constructor Pattern

Delegating construction of objects to the same factory. For example, `addToFront` in `List` was a factory of `Nodes`, `begin` and `end` in Iterator are factories of Iterator

5 Exceptions

Recall the standard vector class.

```
vector<int> v;
v[i];
```

If `i` is not in range, we have undefined behaviour. To fix this, we can do what is called check access.

5.1 Example

```
std::vector<T>::at(int)
v.at(i); // checked access

// client.cc
vector<int> v;
v.at(i);
if (errno) { handle error }
```

5.2 Better Example

We include `<stdexcept>`, and use try-catch blocks

```
void f() {
    cout << "Start f" << endl;
    throw(out_of_range("f"));
    cout << "Finish f" << end;
}

void g() {
```

```
    cout << "Start g" << endl; f(); cout << "Finish g" << endl;
}

void h() {
    cout << "Start h" << endl; g(); cout << "Finish h" << endl;
}

int main() {
    cout << "Start main" << endl;
    try {
        h();
        cout << "Done try" << endl;
    } catch(out_of_range) {cerr << "range error" << endl; }
    cout << "Done main";
}
```

The output is

- Start main
- Start h
- Start g
- Start f
- range error
- Done main

`f` keeps throwing until it finds a try catch block!

5.3 Error Recovery

Error recovery can be done in stages.

5.3.1 Example

```
foo() {
    try {
        // some code
    } catch(SomeError e) {
        // do some part of the recovery
        throw OtherException {...};
    }
}
```

Some ways to handle error recovery:

- Throw a different exception from the catch block
- Throw the exception caught - `throw e`
- Throw the original exception - `throw`
- There is a subtle difference between the last two!

5.4 Exception Classes

- C++ library exceptions inherit from `std::exception`
- We can create exception classes that do not inherit from anything
- C++ allows throwing anything

Advice: Use existing exception classes or create your own

5.4.1 Example

```
class BadInput {};
int n = 0;
try {
    if (!(cin >> n)) throw BadInput();
} catch (BadInput &BI) { // catching by reference avoids slicing and copying
    n = 0;
}
```

5.5 Some types of Exceptions

- `out_range`: thrown by `vector::at`
- `bad_alloc`: new fails to allocate memory
- `length_error`: unable to change length

Destructors should not throw exceptions. WHY?
Default behaviour: If a destructor throws an exception the program calls `std::terminate` and stops.
We can change the default.

```
~Node() noexcept(false) {
    ...
} // we can add exceptions to destructors. Doesn't mean we should
```

An exception as occurred..

- The stack is unwinding (look for catch block)
- Destructors for stack-allocated objects are being called
- The destructor throws an exception

We then have 2 simultaneous uncaught exceptions. `std::terminate` will be called and program quits.

6 Big 5 Revisited

Destructors: Base class destructors are virtual

```
class Book {
    public:
        // Custom big 5 implemented
};

class Text : public Book {
    // Big 5 not implemented
};

Text a{title, author, pages, topic};
Text b{a}; // Will call built-in free copy constructor
```

Default copy constructor:

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

Default/Built-in copy assignment operator

```
a = b // call default Text::operator=

Text &Text::operator=(const Text &other) {
    Book::operator=(other); // superclass
    topic = other.topic;
    return *this;
}
```

Tip: Do superclass in MIL first, then do subclass

Move constructor:

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

This will recompile, but it is incorrect implementation. The object that gets constructed is a copy of the objected being destroyed. This is not as efficent.
We need to treat other as a r-value. How do we resolve this?
C++ has the function `std::move` which tacks an l-value and returns an r-value. Let's implement move constructor again.

```
// Move Constructor
Text::Text(Text &&other):
    Book{std::move(other)},
    topic{std::move(other.topic)} {}

// Move assignment operator
Text &Text::operator=(Text &&other) {
    Book::operator=(std::move(other));
    topic = std::move(other.topic);
    return *this;
}
```

Consider the assignment operator in detail.

```
Text t1{"Mechanics", "Nomair", 50, "Physics"};
Text t2{"C++", "Brad", 500, "CS"};

Book *pt1{*t1};
Book *pt2{&t2};
*pt1 = *pt2 // NOT ptr assignment. This is the assigment operator for objects.
    We can use pointers to achieve object assignment. Which one gets called?
    Book's assignment operator is called because declared type of pt1 is a Book.
```

This is known as the Partial Assignment Problem. What is the solution? Last time we used virtual. **Intent:** Compile finds book's copy constructor, sees virtual, and waits until runtime. At runtime, the right operator is called. But this fails.

```
class Book{
    ...
    virtual Book &operator=(const Book&);
};

class Text {
    ...
    Text &operator = (const Text &) override;
};
```

This fails because override is not an actual override. We need the operator to be able to take any book. This introduces the **Mixed Assignment Pattern**.

```
Text t = ...;
Comic c = ...;
* = c; // How do we get topic from parent class?
```

Alternate solution (sort of)

- Disallow assignment through base class pointers
- Option 1: Make `Make Book::operator=` private
  - Problem: Disallows ALL assignments includin subclass assignments
- Option 2: Make `Make Book::operator=` protected
  - Problem: Disallows assignment of Book objects
- Make abstract base class with protected `operator=`
  - Goal achieved!

```
AbsBook *ap1 = ...; // Wont's compile because = operator is protected
AbsBook *ap2 = ...;
* ap1 = *ap2; // Goal achieved!
```

7 Design Patterns (Again)

7.1 Template Design Pattern

The base class provides some functionality where subclasses may/must provide parts of the behaviour.

```
class Turtle {
    public:
    void draw() { // not virtual, saying this is the only way to draw turtle
        drawHead();
        drawShell();
        drawFeet();
    }
    private:
    void drawHead() {code here}
    void drawFeet() {code here}
    virtual void drawShell() = 0;
};
```

Subclasses fill in the blanks of the base class. In this case, the blank is `drawShell()`.

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

7.2 NVI (non-virtual interface) Idiom

A generalization of the template method pattern. Consider a `public virtual` method.

- `public`: part of the interface, come with pre/post conditions - guarantee certain behaviour
- `virtual`: an invitation to subclasses to change behaviour

In a NVI interface implementation, all `public` methods are non `virtual` (except the destructor), and all `virtual` methods are `private`/`protected`.

We will go through an implemntation with and without NVI.

```
// not NVI
class DigitialMedia { // abstract base class
    public:
    virtual void play() = 0;
    virtual ~DigitalMedia();
};

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

People argue why NVI is better argue that with NVI, the original creator has more control. Without NVI, anyone who makes a subclass has complete control.

7.3 STL: map template class

Parameterized on two types: Key, Value → a dictionary. A generalization of an array/lookup table with one restriction: the Key should support `operator<`.

```
map<string, int> m;
m["abc"] = 5;
m["def"] = 42;
cout << m["def"]; // prints 42
m.erase("def");
if (m.count("abc")) // returns 0 if not found, 1 if found
for (auto &p : m) //p has type std::pair<string, int>{
    cout << p.first << p.second;
}
cout << m["xyz"]; // what happens here? the default value returned (which is 0
    in this case)
```

7.4 Visitor Design Pattern

There are two reasons for this design pattern.

1. Ability to do Double Dispatch

```
virtual void Enemy::strike(Stick &) = 0;
virtual void Enemy::strike(Rock &) = 0;

////////////////////////////////////////

Enemy *e = l->createEnemy();
Weapon *w = player->chooseWeapon();
e->strike(*w);
// the issue is that w is type weapon, but we don't have a strike method
    for a general weapon - dynamic dispatch only occurred on e - C++ and
    other languages do not dynamically dispatch on parameters

////////////////////////////////////////

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

class Turtle : public Enemy {
    void strike(Weapon &w) { w.useOn(*this); } // now the compiler knows
        *this must be a turtle, which is important to do overloading
};

class Bullet : public Enemy {
    void strike(Weapon &w) {}
```

```
};

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

2. Adding functionality to class hierarchy without adding code to the classes. (see lectures/se/visitor)

To summarize, the visitor design pattern uses double dispatch - to figure out both the type of the object and the type of the parameter to end up in the right method.

8 Compilation Dependencies, Design Strategies

8.1 Compilation Dependencies

An `include` cretes a compilation dependency. Ever time an included file changes, we must recompile. Thus, we should avoid including headers as much as much.

- Prefer to forward declare classes such as `class xyz;` instead of including
- A forward declaration of a class is a primise that the type will exist, while the include gives the entire definition

Advantages of forward declaration

- Reduce compilation dependencies (avoid include cycles)
- Fewer compilations
- Faster compile time

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

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

// c.h
#include "a.h"
class C {
```

```
    A a;
};

// d.h
class A;
class D{
    A *myA;
};

// d.cc
#include "a.h"
void D::foo() {
    myA->bar();
}

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

8.2 Design Strategy

8.2.1 pImpl Idiom

The idea behind the pointer to implementation (pImpl) idiom is to take everything that is `private` and put it in another implementation class. For example:

```
// windowImpl.h
#include <Xlib/Xlib.h>
struct XWindowImpl {
    Display *d;
    Window w;
};

// window.h
struct XWindowImpl;
class Window {
    XWindowImpl *pImpl;
    public:
    void drawRectangle();
    void drawString();
};

// window.cc
```

```
#include "XWindowImpl.h"
XWindow::XWindow() : pImpl{new XWindowImpl} {}

XWindow::~XWindow() { delete pImpl; }
```

With this change, client.cc does not need to recompile if the implementation changes.

Generalization: Bridge Design Pattern An extension of the pImpl idiom to accomodate multiple implementations. Recall from CS136 that:

- Coupling: how modules interact with each other - aim for low coupling
- Cohesion: how related are members within a module - aim for high coupling

Decoupling the interface (MVC Design Pattern)
The MVC Design Pattern splits the code into 3 components: the model, view, and controller.

- The Model is the actual data representation (i.e., an array or a vector). For example, a vector of players, cells, or links
- The View is an interface to reading the model. For example, a GUI.
- The Controller handles changing or modifying the data. For example, a game controller that handles the data every turn

Single Responsibility Principle Every class should have only one reason to change.

9 Exception Safety

Consider the following function:

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

Assume g does not leak memory, and the deletion of p does not leak memory either. If g throws an exception, you may still leak memory.

9.1 Exception Safety

Our program should recover from exceptions, have no memory leaks, no dangling pointers, and no broken invariants.

9.2 C++ Guarantee

During stack unwinding, all stack allocated data is destroyed (objects' destructors are run). This is the only guarantee that C++ provides. However, C++ will not automatically call delete on p.

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

The above fix is quite tedious and error-prone, especially for large, nested pieces of code. What do we do when we must use heap memory?

9.3 RAII: Resource Acquisition Is Initialization

- Wrap any resource within a stack-allocated object whose destructor releases the resource
- Heap memory is a resource

Using RAII for heap memory:

```
#include <memory>
std::unique_ptr<T>
```

- constructor takes a T*
- destructor will delete the T*
- overloads for `operator*` and `operator->`

```
void f() {
    std::unique_ptr<MyClass> p{new MyClass};
    MyClass mc;
    g();
}
```

If ownership is shared then we have:

```
std::shared_ptr<T>
shared_ptr<MyClass> p = make_shared<MyClass> ();
if () {
    shared_ptr<MyClass> q = p;
} // q goes out of scope
```

9.4 Three Levels of Exception Safety

- Basic guarantee: If an exception occurs, the program is in a valid bu unspecified state (meaning we do not know how much the program has executed)
- Strong guarantee: If an exception occurs while calling a function `f`, it will be as if `f` was never called
- No throw guarantee: the function will not throw an exception and will achieve its task

It is often difficult to determine what kind of guarantee a function has. For example, assume `method1` and `method2` have a strong guarantee.

```
class A { some code};
class B { some code};
class C {
    A a;
    B b;
    void f () {
        a.method1();
        b.method2();
    }
};
```

What kind of guarantee does `f` have?

- `f` does not have a no throw guarantee, since either of `method1` or `method2` could throw, and `f` would not catch
- Assume `method1` ran and didn't throw. If `method2` throws and we cannot reverse the effects of what `method1` did, then `f` no longer has a strong guarantee.
  - The following example shows that even when we can revert some of the changes back, there is still no strong guarantee

```
void C::f() {
    A aCopy{a};
    B bcopy{b};
```

```
    aCopy.method1();
    bCopy.method2();
    a = aCopy;
    b = bCopy;
}
```

We attempted to write code such that if one of the methods throw, `a` and `b` have not changed. There is still no strong guarantee, since `a = aCopy;` can run correctly but `b = bCopy;` can throw.

9.5 STL: Vector class

`vector` is an example of RAII

```
vector<MyClass> v;
// When the internal array is destroyed, each element is destroyed
//////

vector<MyClass *> v;
// when the array is destroyed, each element (which is a pointer) is destroyed
// NOTE: this does not mean deleting a pointer
// The object this pointer points to still exists!

//////
vector<Observer *> observers;
// In the case where elements are pointers to heap objects ANd the code owns
    these objects, we do:
for (auto &e: v) delete e;

// without using delete:
vector<unique_ptr<MyClass>> v;
vector<T>::emplace_back // has a strong guarantee
```

We can mark methods with a no throw guarantee as follows:

```
class A {
    public:
        void f() noexcept {}
};
```

10 Casting

Casting is the process of forcing one data type to be converted into another. Types of casting:

1. General casting: `(type) expression`

2. static_cast
3. const_cast
4. dynamic_cast
5. reinterpret_cast

10.1 Static Cast

static_cast<Type> (Expression):

- Takes in a type and returns the equivalent value of type Type
- Can be used for non-polymorphic downcasting from a parent pointer to a child pointer
- Does not change the value of variable in expression

10.2 Const Cast

const_cast<Type> (Expression):

- This is used to explicitly override const/ volatile in a cast
- Target type must be the same.
- Used to either set or remove the const attribute

```
void g(int *p);
void f(const int *p) {
    g(const_cast(p));
}
```

10.3 Dynamic Cast

dynamic_cast<Type> (Expression):

- Type-safe form of downcasting by verifying the validity of the cast at runtime
- Performs polymorphic downcasting from a parent pointer to a child pointer
- Returns nullptr if it fails (i.e. pointer does not point to the type specified)
- Safer than static_cast, but less efficient
- Only works on polymorphic classes (have at least one virtual method)

10.4 Reinterpret Cast

`reinterpret_cast<Type> (Expression)`:

- Changes a poiner to any other type of pointer
- Allows casting from pointer to an integer type and vice versa
- This is an implementation-dependent cast and is unsafe

```
Student s;

// Force student to be treated like a turle :(
Turtle *t = reinterpret_cast<Turtle*>(&s);
```

10.5 Downcasting

- Not recommended because we might downcast into a wrong type
- Use polymorphism instead whenever possible
- If we must downcast, use `dynamic_cast` for a type-safe downcast operation (since it returns `nullptr` if cast is not proper

```
Pet *b = new Cat; // upcast
Dog *d1 = dynamic_cast<Dog*>(b); // returns nullptr;
Cat *c1 = dynamic_cast<Cat*>(b); // ok
```

Downcasting References

- This is exception safe because it returns `nullptr` if wrong type
- If we try to `dynamic_cast` a reference variable of the wrong type, `dynamic_cast` will throw a `bad_cast` because there is no such thing as a null reference.

10.6 Run-Time Type Information (RTTI)

RTTI is the ability to provide information about the tye of dynamic-typed object at runtime.

- `dynamic_cast` is a form of RTTI
- We can also Use `typeid` and `static_cast` to downcast.
  - `typeid` returns a reference to a `std::type_info` object which can be used to compare objects/references/pointers with the desired type using `==` and `!=`.

– `typeid(obj).name()` returns a string representation of the type name

```
Cricle c;
Shape *s = &c; //Upcast
if (typeid(Circle) == typeid(s)) s = static_cast<Shape*>(&c);
```

Mechanisms of RTTI with Dynamic Cast When `dynamic_cast<destination*>(src)` is used:

1. Source pointers RTTI info is retrieved
2. RTTI info of destination is fetched
3. Library routine then determine whether source pointers type is of destination type or base class type

Creating Your Own RTTI:
RTTI requires only a virtual function to identify the exact type of the class, and a function to take a pointer to the base type and cast it down to the derived type. This function produces a pointer of the derived type.