

COMP 3522

Object Oriented Programming in C++
Week 12 day 2 Review

Agenda

1. Smart Pointers pt.2

2. Review

1. Move semantics
2. Static vs dynamic memory
3. Functors
4. Lambdas
5. Templates
6. Design patterns
7. lValue, rValue

COMP

3522

SHARED, AND
WEAK POINTERS

shared_ptr

- When all shared_ptrs are out of scope, the memory is deleted (limited garbage collection, again!)
- shared_ptr objects can only share ownership by copying their value
- If two shared_ptr are constructed from the same raw pointer, they will both consider themselves the sole owner
- This can cause potential access problems when one of them deletes its managed object and leaves the other pointing to an invalid location
- Check out [**shared_ptr_1.cpp**](#), [**shared_ptr_2.cpp**](#)

An important fact about make_shared

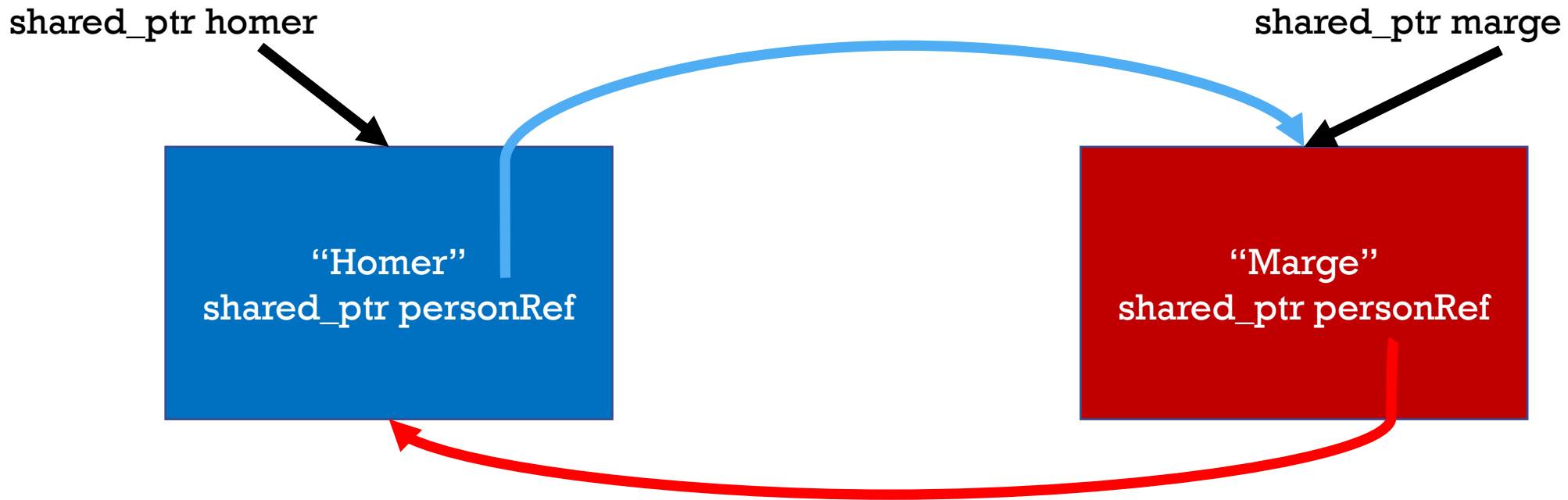
Though it is possible to create a shared_ptr by passing a pointer to its constructor, **constructing a shared_ptr with make_shared should be always preferred.**

It is **more efficient** (only requiring one memory allocation rather than two).

3. weak_ptr

- Holds a non-owning reference to a pointer managed by shared_ptr
- Must be converted to a shared_ptr to access the object
- Models temporary ownership
- Check out [weak_ptr_1.cpp](#), **[weak_ptr_2.cpp](#)**, [weak_ptr_3.cpp](#), [weak_ptr_4.cpp](#)

3. weak_ptr circular reference



- Primarily used in rare cases to break circular references, i.e., in doubly linked lists
- Homer and Marge's personRef shared pointer has reference count 2
- Can't call destructor on shared pointers if reference count > 1

Smart pointer guidelines

- When an object is dynamically allocated, immediately assign it to a smart pointer that will act as its 'owner'
- If a program will need more than one pointer to an object, use `shared_ptr`
- If a program doesn't need multiple pointers to the same object, use a `unique_ptr`
- Final word: check out [**code_snippet_1.cpp**](#)

REVIEW

Information

- Final: Tuesday Dec 10, 8am-10am @ GYM
- Focus on material after midterm 1
 - May indirectly touch material from before midterm
 - Ie: design patterns require knowledge of inheritance, polymorphism, classes, interfaces, abstract classes etc
- Go over slides. Understand code samples
- Complete sample final questions
- Bring 1 page – hand written single sided cheat sheet to exam
 - Hand it in with your final

MOVE SEMANTICS

Move semantics

- When mentioning “Move constructor” generally referring to constructor that moves data from parameter object to this object
- Kind of like copy constructor, but without the copying. Moving instead

```
myClass (myClass&& mc) {  
    //moves data from o to this class  
}
```

Move semantics

- Move assignment looks like this

```
myClass& operator=(myClass&& mc) {  
    //moves data from o to this class  
    return *this  
}
```

- It's possible to have a 'regular' constructor that takes in parameters by rvalue

```
myClass(string &&name) : name(move(name)) {  
    cout << "move 'regular' constructor" << endl;  
}
```

STATIC VS DYNAMIC MEMORY

Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object name{};  
}
```

```
//main.cpp  
createObject();  
//some other code
```

- Let's look at statically allocated memory first
- Create object inside of function

Object *name = new Object() vs Object name{

```
void createObject() {  
    Object name{};  
}
```

```
//main.cpp  
createObject();  
//some other code
```

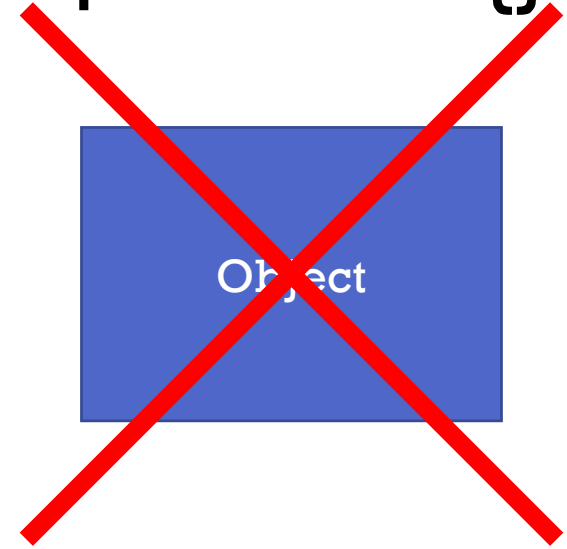


Create object in memory

Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object name{};  
}
```

```
//main.cpp  
createObject();  
//some other code
```



After leaving createObject function, object is released from memory

Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```

- Now look at dynamically allocated memory

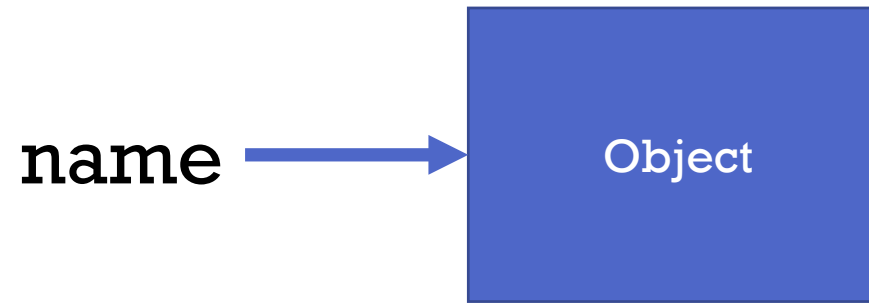
Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp
```

```
createObject();
```

```
//some other code
```



- Create object in memory.
- name pointer points to object

Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```

~~name~~



After leaving function, pointer memory is released, but object still exists in memory

Object *name = new Object() vs Object name{}

```
void createObject() {  
    Object *name = new Object;  
}
```

```
//main.cpp  
createObject();  
//some other code
```



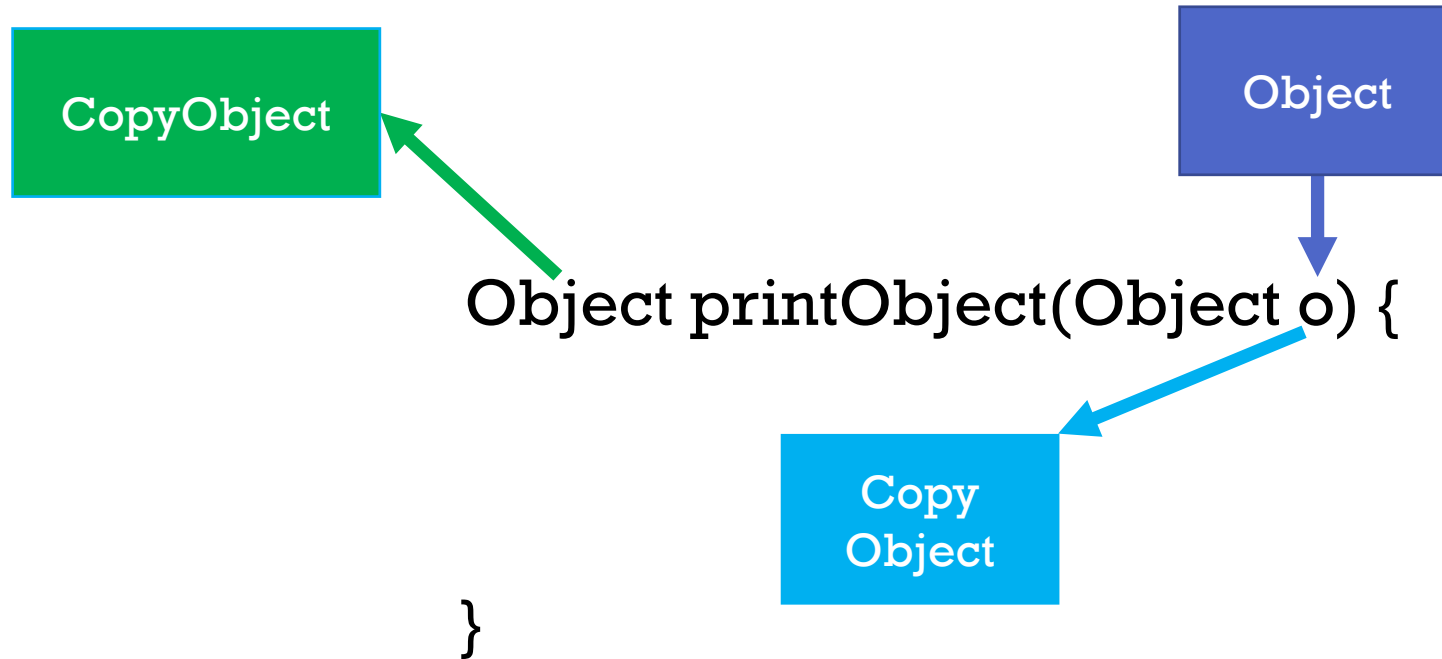
- This is a memory leak because there's no pointer pointing to object
- No way to call delete on the object

Why use dynamic memory?

- There are situations where you want memory to persist beyond the scope of where the memory is created
- This is where you use dynamic memory
- It's more efficient than statically allocating memory and copying that memory everywhere you need it
- Cases where you don't know how much memory you'll need at run-time
 - Think of the linked list lab
 - Don't know ahead of time how long the linked list is
 - Need the linked list to dynamically add new nodes

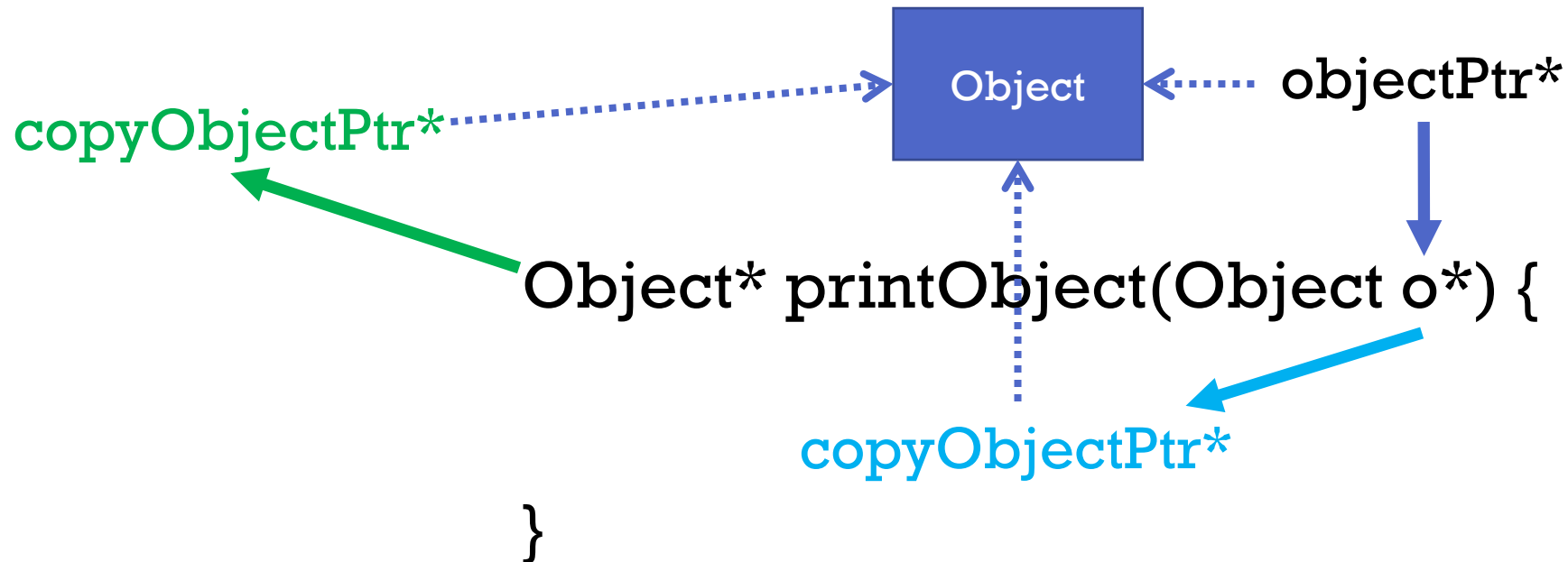
Pass by value

- This shows calling a function by passing the object
- Look at all the copied objects
 - Expensive operation and memory allocation



Pass by value

- This shows calling a function with object pointer
- Only one object is pointed at, no object copies made
- Only the pointers are copied, this is fast compared to copying object



FUNCTION OBJECTS

The C++ function object

- It is a generalization of a function
- Useful as predicates or comparison function (STL algorithms)
- Often called a **functor**
- It is an **object that acts like a function**
- **Accomplished by overloading the parentheses operator**

```
F f;  
f(1); // like f.operator()(1);
```

A simple example from cplusplus.com

```
struct myclass {  
    int operator()(int a) { return a;}  
} myobject;  
  
// Looks like a function call, so much neat!  
int x = myobject(0);
```

Why use Functors?

- Let's compare functions and functors
- Functions don't maintain internal state
 - Ie: After calling a function, any local variables inside of the function are gone
- Functors are actual objects, meaning they contain state
 - Can have member variables that exist beyond the scope of member functions

Why use Functors?

```
struct myFunctor {  
    int sum;  
    void operator()(int x)  
{  
        sum += x;  
    }  
};
```

```
myFunctor addNum, addNum2;
```

```
addNum(5);  
addNum(10);  
//addNum sum = 15
```

```
addNum2(100);  
addNum2(200);  
//addNum2 sum = 300
```

LAMBDA

What is a lambda expression?

- An **unnamed function object** (functor)
- A form of “**nested function**”
- You should use lambda expressions when the “function” is used a limited number of times
- Java lingo reminder: it's one of those anonymous things!

The general form of a lambda expression

```
[capture clause] (parameters)<specifiers> -> <return type>
{
    body
}
```


Capture clause

- Used to pass variables from the surrounding scope into the lambda expression:
 1. **[]** empty there is no capturing
 2. **[=]** outside variables are captured by value and cannot be modified inside the lambda expression
 3. **[&]** outside variables are captured by reference
 4. **[variable_name]** only variable_name is captured by value and cannot be modified inside the lambda
 5. **[&variable_name]** only variable_name is captured by reference

When should we lambda?

Lambdas are great for:

1. Short one-lined functions
2. Functions passed to STL containers, i.e., something to compare elements in a queue
3. Functions that are used in just one place.

Compare functor (function objects) vs lambda

```
class increment
{
private:
    int num;
public:
    increment(int n) : num(n) {}
    int operator () (int arr_num) const {
        return num + arr_num;
    }
};

vector<int> v = {1, 2, 3, 4, 5};
int add = 5;
transform(v.begin(), v.end(), v.begin(), increment(add));
```

Compare functor (function objects) vs lambda

```
vector<int> v = {1, 2, 3, 4, 5};  
int add = 5;  
  
transform(v.begin(), v.end(), v.begin(), [add](int arr_num)  
{  
    return add + arr_num;  
});
```

TEMPLATE PROGRAMMING

Template programming

- Create generic code that can be used with any data type
- Add template <typename T> before a function or class
 - Any references to T refers to a eventual concrete type passed into to the function or class
- Keyword **typename** is interchangeable with **class**

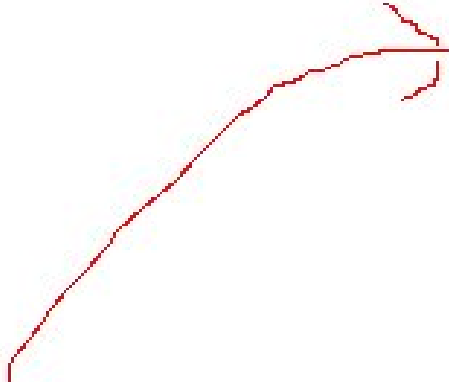
Template programming: Template function

Compiler internally generates and adds below code

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}
```


```
int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

int myMax(int x, int y)
{
 return (x > y)? x: y;
}



Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```



Function template type inference

```
template <typename T, typename T2>
void myPrint2(T t, T2 t2) {
    cout << t << " " << t2 << endl;
}

//main function
myPrint2(10, 'b');
```

- Compiler can infer 10 is an int, 'b' is char
- First parameter 10 goes into T, 'b' goes into T2

Function template type inference

```
template <typename T>
void myPrint1(T t, T t2) {
    cout << t << " " << t2 << endl;
}

//main function
myPrint2(10, 'b'); //ERROR
```

- Compare to this function, that only has 1 template parameter T
- Compiler expects both t and t2 to be the same type T
- But passing in an int (10) and char ('b')

Function template type inference

```
template <typename T>
void myPrint1(T t, T t2) {
    cout << t << " " << t2 << endl;
}

//main function
myPrint2<int>(10, 'b'); //OK
```

- Need to add <int> to explicitly indicate both parameters are to be interpreted as int
- The output is 10, 98 //98 is ascii for 'b'

Function template type inference

```
template <typename T, typename T2>
void myPrint2(T t, T2 t2, T2 t3) {
    cout << t << " " << t2 << " " << t3 << endl;
}
```

//main function

```
myPrint2(10, 'b', 'b'); //OK (T, T2, T2)
```

```
myPrint2(10, 'b', 10); //ERROR (T, T2, T)
```

```
myPrint2(10, 10, 'b'); //ERROR (T, T, T2)
```

```
myPrint2(10, 'b', "hello"); //ERROR (T, T2, T?)
```

- Templates expects 2 possible types T, T2
- Parameters must be in the type order (T, T2, T2)

Class template friend functions. Option 1

```
template <typename T>                                //definition outside class
class A {
    T t;
public:
    A(T t) : t(t) {}

    template<class T> //ERROR
    friend A<T> foo(A<T>& a);
};

A<T> foo(A<T>& a)
{
    cout << "foo" << endl;
    return a;
}
```

- Template parameter **T** above foo function hides **T** of class template parameter

Class template friend functions. Option 1

```
template <typename T>                                //definition outside class
class A {
    T t;
public:
    A(T t) : t(t) {}

    template<class U> //OK!
    friend A<U> foo(A<U>& a);
};

template<class T>
A<T> foo(A<T>& a)
{
    cout << "foo" << endl;
    return a;
}
```

- Solution 1: Change template parameter above foo function to **U** or any other unused parameter name

Class template friend functions. Option 2

```
template <typename T>
class A {
    T t;
public:
    A(T t) : t(t) {}

    friend A<T> foo2(A<T>& a) {
        cout << "foo2" << endl;
        return a;
    }
};
```

- Solution 2
- Write a friend function with without “**template** <typename T>” on top of function
- Complete definition of function inside class

DESIGN PATTERNS

Singleton

- Globally accessible class
- Guarantees only one instance is created
- Advantage
 - Reduces coupling
 - Ease of access to data
- Disadvantage
 - Easy to write badly designed code
 - Since it's globally available, an unfocused singleton will be called by many places in code

Abstract factory

- Create objects using abstraction instead of concrete classes
- Advantage
 - Easy to add new products and factories without touching creation logic
 - Follows open closed principle
 - Avoid tight coupling between products and client
- Disadvantage
 - Lots of coding and classes to add functionality

Observer

- Subject broadcasts information to N number of observers. Observers react based on new information
- Advantage
 - Open/Closed principle. New subscribers can be added to subject without subject needing to change their code
 - Can add new observers at runtime
- Disadvantage
 - Order that observers are notified is based on the order observers are attached to subject

Decorator

- Add functionality to object at runtime
- Advantage
 - Extend object functionality without making new subclasses
 - Can add/remove functionality at runtime
- Disadvantage
 - Difficult to remove specific functionality from an object
 - Design pattern itself is difficult to understand
 - Difficult to implement decorator where its functionality doesn't depend on the order in which it's called

Facade

- Provides an interface to simplify complex class functionality from client
- Advantage
 - Separate client from complex subsystems
- Disadvantage
 - Easy to have a façade highly coupled with all subsystems

Proxy

- Client uses a service as regular. But in reality they're using a proxy version of the service
- Advantage
 - Control service without client knowledge
 - Open/Closed principle. Can introduce new proxies without changing service or clients
 - Proxy can add functionality on top of service without modifying service
- Disadvantage
 - Complicated code because of new interfaces/classes
 - Delayed response from service

Strategy

- Extract functionality of a class into separate classes called strategies. Object can add/remove functionality at runtime
- Advantage
 - Swap functionality at runtime
 - Replace inheritance with composition
 - Isolate implementation from code that uses it
 - Open/Closed principle: Can add new strategy without changing calling class
- Disadvantage
 - Client needs to know of strategies to use the correct one
 - No need to over complicate code if there are only a few strategies

LVALUE & RVALUE

lvalue and rvalue

- Before move semantics
 - lvalue – expression on the left assignment operator
 - rvalue – expression on the right of assignment operator
- After move semantics
 - lvalue – expression which has a memory address
 - rvalue – expression which does NOT have a memory address
 - `int y = 0;` //y is lvalue, 0 is rvalue
 - `int x = y;` //x is lvalue, y also lvalue
- Easy way to know, can you use the address of operator(&) on an expression?
 - If yes, then that expression is an lvalue `//cout << &x; //OK!`
 - If no, then it's an rvalue `//cout << &0; //ERROR!`

Thank you for being a great class!

Survey time!