
Please do not redistribute slides without
prior permission.



Cppcon
The C++ Conference

2020 
September 13-18
ONLINE
GOING VIRTUAL

Back to Basics: Design Patterns

Mike Shah, Ph.D.

[@MichaelShah](#) | mshah.io

September 17, 2020

60 minutes | Introductory to Intermediate Level Audience

This talk is being delivered to a live audience!

- I'll take questions at the end in the online conference hall
 - (Back to Basics Room 1 at Table 1)
- I look forward to discussion!

Abstract

The abstract that you read and enticed you to join me is here!

Design Patterns are reusable elements of design that may help aid in making software more maintainable, flexible, and extensible. The term 'design patterns' can be traced back to at least the 1970s, although the term has been largely popularized by the 'Gang of Four' book Design Patterns, in which common software design patterns were defined and categorized. In this talk, you will learn the fundamentals of the creational, structural, and behavior design patterns. This talk is aimed at beginners who have some C++ knowledge working on a software project, but are starting to think about larger software problems. This talk will also be useful for folks who have been working in C++ for a while, but have never had a chance to study design patterns and need some resources to help orient them.

Learning about design patterns and where to apply them can at the least give you a way to think about how you solve unknown problems, or otherwise organize your software--think about design patterns as another tool to add to your developer toolbox. We will start this talk by introducing the taxonomy of design patterns at a high level, how to read a UML diagram (as a quick refresher), a refresher on inheritance vs composition, and then spend the rest of the time on walking through the implementation of several design patterns. Attendees will leave this talk ready to implement and use design patterns in C++.

Who Am I?

by Mike Shah

- **Assistant Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I teach courses in computer systems, computer graphics, and game engine development.
 - My **research** in program analysis is related to **performance** building static/dynamic analysis and software visualization tools.
- I do **consulting** and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
 - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of **computer graphics**, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io



What you are going to learn today

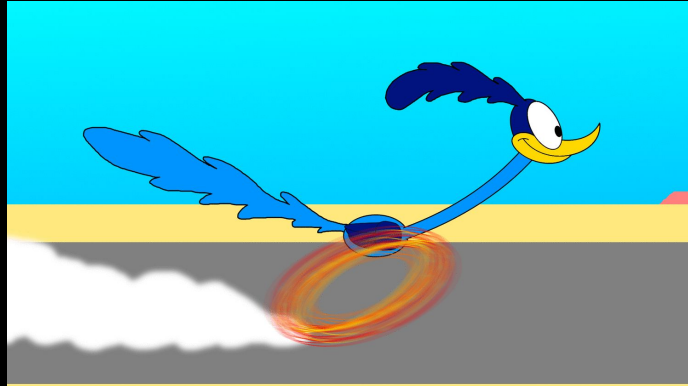
- (So you know what to pay attention to)
- What Design Patterns are and why you should study them
- Pragmatically
 - How to implement/use a Creational Pattern
 - Singleton in C++
 - How to implement/use a Structural Pattern
 - Adapter in C++
 - How to use a Behavior Pattern
 - Iterator in C++

What you are going to learn today

- (So you know what to pay attention to)
- What Design Patterns are and why you should study them
- Pragmatically
 - How to implement/use a Creational Pattern
 - Singleton in C++
 - How to implement/use a Structural Pattern
 - Adapter in C++
 - How to use a Behavior Pattern
 - Iterator in C++
- My expectations are you:
 - Have written some C++ (understand memory allocation, object lifetime)
 - Have used some object-oriented constructs and understand their definitions (understand classes, structs, encapsulation, polymorphism, inheritance, composition)

Okay--let's start building software!

Okay--let's start building software!



So here's the deal

- You are hired as a programmer to perform some maintenance
 - The software is a very exciting screen saver software!
 - You're going to get lots of \$\$\$ to do a good job
 - You're going to be working on this project with a large team
 - If you do a good job, you'll get hired to do another more lucrative contract
 - Sounds good?
 - Sign the dotted line and let's begin!



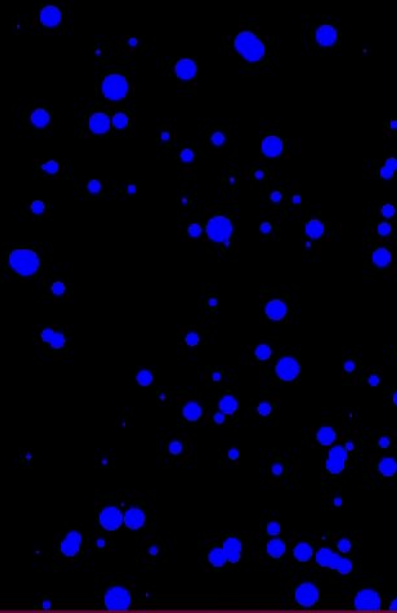
Your first day of work



<https://i.insider.com/59b7d6ec24884943801f84f9?width=600&format=jpeg&auto=webp>

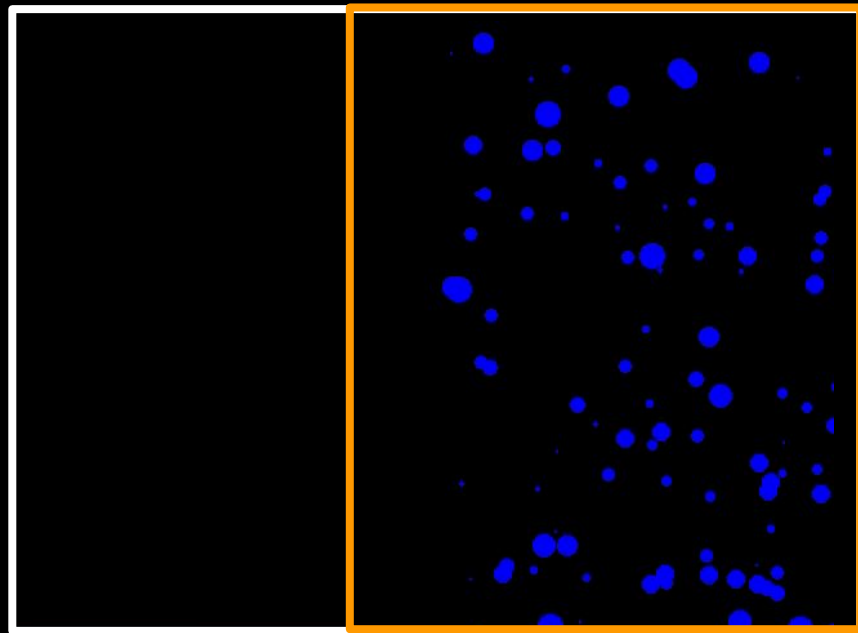
So here's what you're given!

- It's a screensaver with a bunch of circles bouncing around.
 - Your job is to maintain and improve this software
- The initial effect is visually appealing, but quickly you notice a few subtle errors.



So here's what you're given!

- It's a screensaver with a bunch of circles bouncing around.
 - Your job is to maintain and improve this software
- The initial effect is visually appealing, but quickly you notice a few subtle errors.
 - E.g.
 - All of the circles start out in a fixed portion of the screen
 - This is a bug!



Bug Hunting (1/2)

- So you have spotted the bug that generates the starting position of each circle
 - The `Circle` constructor generates a random starting 2D position of:
 - `[1,300]` for `x`
 - `[1,300]` for `y`

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%300 + 1;
12    m_y = rand()%300 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

Bug Hunting (2/2)

- So you have spotted the bug that generates the starting position of each circle
 - The Circle constructor generates a random starting 2D position of:
 - [1,300] for x
 - [1,300] for y
- You observe however, the window size is actually 640x480 stored in two separate global variables
 - So the range should be much larger for the initial circles to spawn in the screen saver

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%300 + 1;
12    m_y = rand()%300 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

First attempt at fixing the bug

- Here's the quick fix (because a deadline is coming up)

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%300 + 1;
12    m_y = rand()%300 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```


First attempt at fixing the bug

- Here's the quick fix (because a deadline is coming up)
 - Just change the range manually!
 - Problem solved!

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%640 + 1;
12    m_y = rand()%480 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

But a problem remains (1/3)

- But the bug is merely hidden, waiting to resurface again, because you have to maintain the value in two different places

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%640 + 1;
12    m_y = rand()%480 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

But a problem remains (2/3)

- But the bug is merely hidden, waiting to resurface again, because you have to maintain the value in two different places
 - Well--the point of having a global variable is to use it after all!
 - How about we just move it to a header file, and define some extern's...
 - Yikes, this is getting nasty!
 - (Now we are getting the linker involved with symbol resolution)
 - Surely a solution must exist to this common problem
 - So there has to be a better way...

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%640 + 1;
12    m_y = rand()%480 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

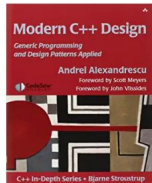
But a problem remains (3/3)

- But the bug is merely hidden, waiting to resurface again, because you have to maintain the value
 - Well--the point of use it after all!
 - How about we just define some external
 - Yikes, this is
 - (Now we are getting the linker involved with symbol resolution)
 - Surely a solution must exist to this common problem
 - So there has to be a better way...

Introducing--Design Patterns!

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%640 + 1;
12    m_y = rand()%480 + 1;
13    // Set the shapes radius
14    setPosition(m_x,m_y);
15    setRadius(m_radius);
16    setFillColor(sf::Color::Blue);
17    setOrigin(m_radius,m_radius);
```

```
18
19 // Library
20 #include <SFML/Window.hpp>
21 #include <SFML/Graphics.hpp>
22
23
24
25
26 // Our includes
27 #include "circle.hpp"
28
29 // C++ Standard Template Library
30 #include <vector>
31 #include <ctime>
32
33 constexpr int windowHeight{640};
34 constexpr int windowHeight{480};
```



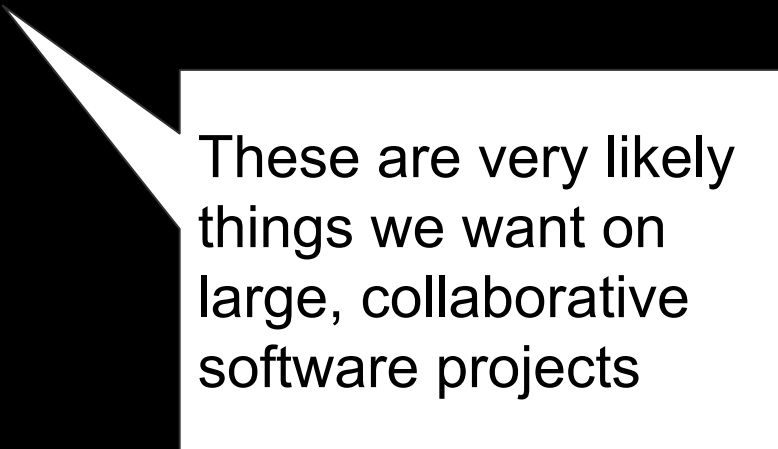
Design Patterns

Design Pattern (1/2)

- A common repeatable solution for solving problems.
 - Thus, Design Patterns can serve as 'templates' or 'flexible blueprints' for developing software.
- Design patterns can help make programs more:
 - Flexible
 - Maintainable
 - Extensible

Design Pattern (2/2)

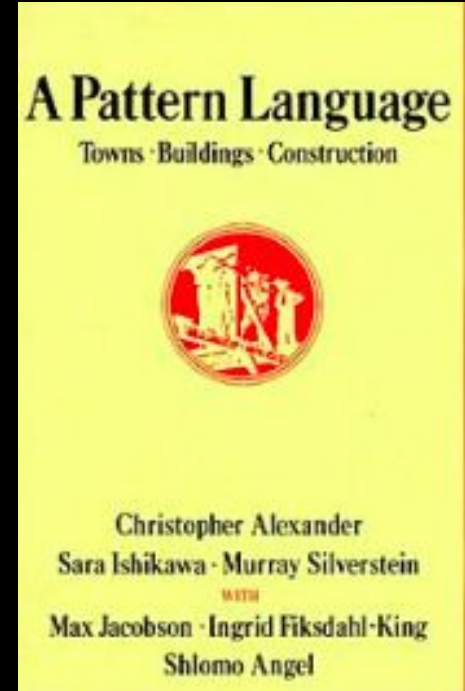
- A common repeatable solution for solving problems.
 - Thus, Design Patterns can serve as ‘templates’ or ‘flexible blueprints’ for developing software.
- Design patterns can help make programs more:
 - Flexible
 - Maintainable
 - Extensible



These are very likely things we want on large, collaborative software projects

Origins of Design Patterns

- Origin of the term “design patterns” in Software Engineering was inspired by reusable elements of design (“patterns”) in the field of architecture
- 1977 book “A Pattern Language: Towns, Buildings, Construction” by Christopher Alexander et al. presents 253 patterns, covering advice on use of materials, physical arrangements of architectural elements, etc.



Design Patterns Book

- In 1994 a book came out collecting heavily used patterns in industry titled “Design Patterns”
 - It had four authors, and is dubbed the “Gang of Four” book (GoF).
 - It is popular enough to have a wikipedia page:
https://en.wikipedia.org/wiki/Design_Patterns
 - C++ code samples included, but can be applied in many languages.
 - This is a good starting point on design patterns for object-oriented programming



Design Patterns Book * Brief Aside *

- In 1994 a book came out collecting heavily used patterns in industry titled “Design Patterns”



- I really enjoyed this book (as a graphics programmer) for learning design patterns.
 - There's a free web version here: <https://gameprogrammingpatterns.com/>
 - I also bought a physical copy to keep on my desk

Design Patterns
Elements of Reusable
Object-Oriented Software

ADDITIONALITY PROFESSIONAL COMPUTING SERIES



Design Patterns Book

- So design patterns are reusable templates that can help us solve problems that occur in software
 - One (of the many) *nice* thing the Design Patterns Gang of Four (GoF) book does is organize the 23* presented patterns into three categories:
 - Creational
 - Structural
 - Behavioral



*Keep in mind there are more than 23 design patterns in the world

Design Pattern Categories (1/3)

There are '3' categories of design patterns

1. Creational

- Provide program more flexibility on how to create objects, often avoiding direct instantiation of a specific object.

2. Structural

- Focus on using inheritance to compose interfaces and define objects in a way to create new functionality.

3. Behavioral

- Focuses on how to communicate between objects

Creational [\[edit \]](#)

Main article: Creational pattern

Creational patterns are ones that create

- Abstract factory groups object facto
- Builder constructs complex objects
- Factory method creates objects with
- Prototype creates objects by cloning
- Singleton restricts object creation to

Structural [\[edit \]](#)

These concern class and object compo

- Adapter allows classes with incomp
- Bridge decouples an abstraction fro
- Composite composes zero-or-more
- Decorator dynamically adds/overrid
- Facade provides a simplified interfa
- Flyweight reduces the cost of creati
- Proxy provides a placeholder for an

Behavioral [\[edit \]](#)

Most of these design patterns are spec

- Chain of responsibility delegates o
- Command creates objects which en
- Interpreter implements a specializ
- Iterator accesses the elements of a
- Mediator allows loose coupling betw
- Memento provides the ability to res
- Observer is a publish/subscribe pat
- State allows an object to alter its be
- Strategy allows one of a family of al
- Template method defines the skelet
- Visitor separates an algorithm from

Design Pattern Categories (2/3)

There are '3' categories of design patterns

1. Creational

- Provide direct i

2. Structural

- Focus

3. Behavioral

- Focuses on how to communicate between objects

I'm going to help us pick the first pattern to solve our original problem (with the screensaver)

Creational [edit]

Main article: *Creational pattern*

Creational patterns are ones that create

- [Abstract factory](#) groups object facto
- [Builder](#) constructs complex objects.
- [Factory method](#) creates objects with
- [Prototype](#) creates objects by cloning
- [Singleton](#) restricts object creation fo

Structural [edit]

These concern class and object compo

- [Adapter](#) allows classes with incom
- [Bridge](#) decouples an abstraction fro
- [Composite](#) composes zero-or-more
- [Decorator](#) dynamically adds/overrid
- [Facade](#) provides a simplified interfa
- [Flyweight](#) reduces the cost of creati
- [Proxy](#) provides a placeholder for an

Behavioral [edit]

Most of these design patterns are speci

- [Chain of responsibility](#) delegates co
- [Command](#) creates objects which en
- [Interpreter](#) implements a specializ
- [Iterator](#) accesses the elements of a
- [Mediator](#) allows [loose coupling](#) betw
- [Memento](#) provides the ability to rest
- [Observer](#) is a publish/subscribe pat
- [State](#) allows an object to alter its be
- [Strategy](#) allows one of a family of al
- [Template method](#) defines the skelet
- [Visitor](#) separates an algorithm from

Design Pattern Categories (3/3)

There are '3' categories of design patterns

1. Creational

- Provide direct i

2. Structural

- Focus way to create new functionality.

3. Behavioral

- Focuses on how to communicate between objects

And it happens to be a creational design pattern

Creational [edit]

Main article: [Creational pattern](#)

Creational patterns are ones that create

- [Abstract factory](#) groups object facto
- [Builder](#) constructs complex objects.
- [Factory method](#) creates objects with
- [Prototype](#) creates objects by cloning
- [Singleton](#) restricts object creation fo

Structural [edit]

These concern class and object compo

- [Adapter](#) allows classes with incom
- [Bridge](#) decouples an abstraction fro
- [Composite](#) composes zero-or-more
- [Decorator](#) dynamically adds/overrid
- [Facade](#) provides a simplified interfa
- [Flyweight](#) reduces the cost of creati
- [Proxy](#) provides a placeholder for an

Behavioral [edit]

Most of these design patterns are speci

- [Chain of responsibility](#) delegates co
- [Command](#) creates objects which en
- [Interpreter](#) implements a specializ
- [Iterator](#) accesses the elements of a
- [Mediator](#) allows [loose coupling](#) betw
- [Memento](#) provides the ability to rest
- [Observer](#) is a publish/subscribe pat
- [State](#) allows an object to alter its be
- [Strategy](#) allows one of a family of al
- [Template method](#) defines the skelet
- [Visitor](#) separates an algorithm from

Creational Design Patterns

- Creational design patterns deal with how objects are instantiated--often abstracting the process
 - Typically this means encapsulating how an object is created
 - (i.e. How, or what type of object is being allocated when you use the `new` operator)

Problem we are solving

- We want to be able to reference `windowWidth` and `windowHeight` globally
 - Ideally we only want one instance of these values to keep track of
 - We also would prefer not to maintain individual global variables, especially if they are related
 - (e.g. we may use `windowWidth` and `windowHeight` to compute an aspect ratio)

```
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9
10    // Set the initial starting point
11    m_x = rand()%640 + 1;
12    m_y = rand()%480 + 1;
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

```
1 // main.cpp
2 // Third-Party Library
3 #include <SFML/Window.hpp>
4 #include <SFML/Graphics.hpp>
5
6 // Our includes
7 #include "circle.hpp"
8
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
```

Singleton

A Creational Design Pattern

Singleton

Enforces the existence of only one instance of an object being created

This means we can store all of our configuration values in exactly one place

Singleton Pattern (1/3)

- **Pattern:** Enforces the existence of only one instance of an object being created and have it be globally accessible

Singleton Pattern (2/3)

- **Pattern:** Enforces the existence of only one instance of an object being created and have it be globally accessible
- **Common Uses:**
 - Managing files
 - (i.e. a File System)
 - Resource Managers
 - (i.e. Managing specific types of data)
 - A Logger
 - (i.e. Log messages or errors in a centralized database)
 - A configuration manager
 - Hold global configuration value

Singleton Pattern (3/3)

- **Pattern:** Enforces the existence of only one instance of an object being created and have it be globally accessible
- **Common Uses:**
 - Managing files
 - (i.e. a File System)
 - Resource Managers
 - (i.e. Managing specific types of data)
 - A Logger
 - (i.e. Log messages or errors in a centralized database)
 - **A configuration manager**
 - **Hold global configuration values**



This is the option we want!

Time for the refactoring! (1/2)

- The first step is to move our global variables into a class
 - This groups both variables together
 - We will likely want to also have some behaviors (i.e. member functions) for working with these values as well


```
9 // C++ Standard Template Library
10 #include <vector>
11 #include <ctime>
12
13 constexpr int windowHeight{640};
14 constexpr int windowHeight{480};
15
16
17
18
```

Time for the refactoring! (2/2)

- We move our globals into a class and will make them member variables
 - ConfigurationManager will be our Singleton
 - Through ConfigurationManager we will access and share data,

```
13 constexpr int windowHeight{640};
14 constexpr int windowWidth{480};

1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager* instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructors
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```



Closer look at the C++ (1/5)

We have member variables storing the information we need in our class.

```
1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager& instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructor
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```


Closer look at the C++ (2/5)

In order to access our member variables, there are corresponding 'getter' and 'setter' member functions

```
1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager& instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructor
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```

Closer look at the C++ (3/5)

The real interesting part is here:

This is where we have a 'static' member function.

Remember, 'static' makes this function shared across **all** instances of ConfigurationManager's

```
1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager& instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructor
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```

Closer look at the C++ (4/5)

And we are using 'instance' to access a single instance that is only allocated once.

The same instance that is allocated the first time will always be returned.

(In C++11 and beyond, static locals will only be initialized once)

```
1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager& instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructor
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```

Closer look at the C++ (5/5)

We have now encapsulated how an object is created, and also restricted creation to just 1 object of this type.

```
1 #ifndef CONFIGURATION_MANAGER_H
2 #define CONFIGURATION_MANAGER_H
3
4 class ConfigurationManager{
5     public:
6         static ConfigurationManager& instance(){
7             static ConfigurationManager* instance = new ConfigurationManager();
8             return *instance;
9         }
10        // Getter functions
11        int getWindowWidth();
12        int getWindowHeight();
13        // Setter functions
14        void setWindowWidth(int w);
15        void setWindowHeight(int h);
16    private:
17        // Hidden Constructor
18        ConfigurationManager();
19        ~ConfigurationManager();
20        // Member variables
21        int m_windowWidth;
22        int m_windowHeight;
23 };
24 #endif
```

Utilizing the Singleton (1/3)

- Here is an example of accessing our Singleton through the 'instance' static member function
 - And because we are always accessing information through instance(), all of the data will be the same!

```
21 ConfigurationManager::instance().setWindowWidth(640);
22 ConfigurationManager::instance().setWindowHeight(480);
23
24 // Create a window to draw graphics on
25 sf::RenderWindow window{sf::VideoMode(ConfigurationManager::instance().getWindowWidth(),
26                                     ConfigurationManager::instance().getWindowHeight(),
27                                     32),
28                          "Screen Saver - Singleton!"};
```

Utilizing the Singleton (2/3)

- Here is an example of accessing our Singleton through the 'instance' static member function
 - And because we are always accessing information through instance(), all of the data will be the same!

```
2 #include "ConfigurationManager.hpp"
3
4 Circle::Circle(){
5     m_up = (int) (rand()%2 +0 );
6     m_left = (int) (rand()%2 +0);
7     m_radius = (int) (rand()%10 + 1);
8
9     // Set the initial starting point
10    m_x = rand()%ConfigurationManager::instance().getWindowWidth()+ 1;
11    m_y = rand()%ConfigurationManager::instance().getWindowHeight() + 1;
12
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

Utilizing the Singleton (3/3)

- Here is an example of accessing our Singleton through the 'instance' static member function
 - And because the data will be the same

Let's see if this solved our problem

```
7     m_radius = (int) (rand()%10 + 1);
8
9     // Set the initial starting point
10    m_x = rand()%ConfigurationManager::instance().getWindowWidth()+ 1;
11    m_y = rand()%ConfigurationManager::instance().getWindowHeight() + 1;
12
13    // Set the shapes radius
14    m_shape.setPosition(m_x,m_y);
15    m_shape.setRadius(m_radius);
16    m_shape.setFillColor(sf::Color::Blue);
17    m_shape.setOrigin(m_radius,m_radius);
18 }
```

Success!

- If you watch this animation from the start, you'll notice we have a circles that spawn the whole range of the screen
- Ah--much better!

Edit View Search Terminal Help

```
ke:screensaver$ Here we go! in 3..2..1
```

Wrapping up the Singleton

Singleton: Pros and Cons

- Pros

- Easy overall implementation
- Effective when you know you only need one of something
- Avoids polluting global namespace with lots of variables
- Memory is only allocated if you actually use the Singleton

- Cons

- Wrong usage could take refactoring
 - (i.e. In the case that you need two instances of an object)
- Not thread-safe, you will have to add a lock, which may slow down application
 - Hard to tell which thread last modified singleton
- You are still effectively using globals

Singleton: Missing Details

- The Gang of Four book provides much more detail into the Singleton
- There are some design considerations to think of as well:
 - In my code for example, we may want to also hide the copy constructor to make it non-copyable
 - You'll also want to think about if a client can subclass from your Singleton or not.
 - Thread-safety may be an issue
 - You can address this with locks or other atomic data structures depending on your need
 - If you do need multiple instances of a class, but they share some data, consider the 'monostate' pattern
 - i.e. use static member variables
 - Be careful if you return a 'pointer' versus a reference from your instance() function
 - A client may delete your Singleton--you can disallow that if you like

More on the Singleton Pattern - C++

- See the Game Programming Patterns book for some other use cases:
 - <https://gameprogrammingpatterns.com/singleton.html>

```
class FileSystem
{
public:
    static FileSystem& instance()
    {
        static FileSystem *instance = new FileSystem();
        return *instance;
    }

private:
    FileSystem() {}
};
```

Singleton: Missing Details

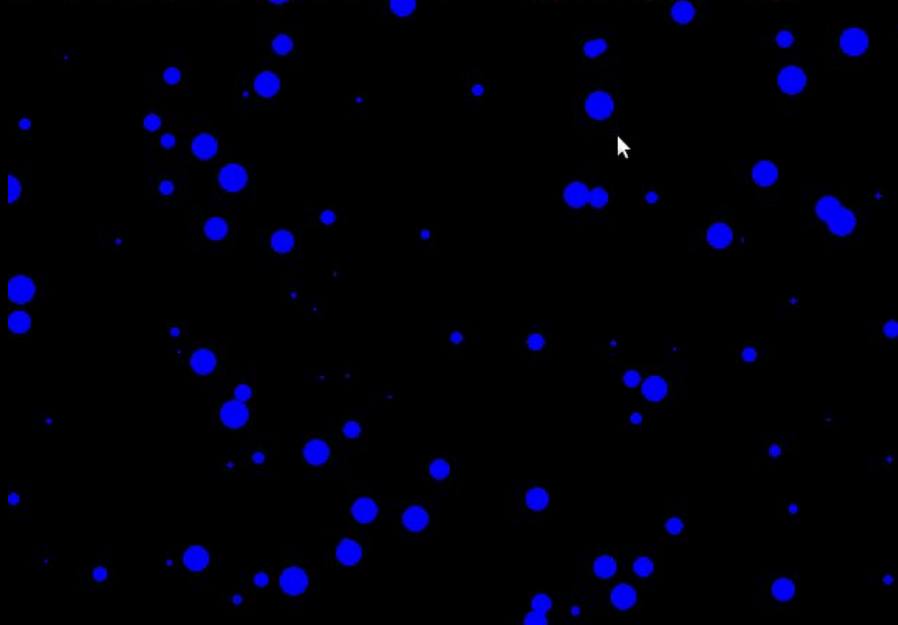
- The Gang of Four book provides much more detail into the Singleton
- There are some design considerations to think of as well:
 - In my code for example, we may want to also hide the copy constructor to make it non-copyable
 - You'll also want to make sure the Singleton is thread-safe
 - Thread-safety is a challenge
 - You can use a mutex to protect the Singleton instance, but this may not be the best solution
 - If you do need multiple instances of a class, but they share some data, consider the 'monostate' pattern
 - i.e. use static member variables
 - Be careful if you return a 'pointer' versus a reference from your instance() function
 - A client may delete your Singleton--you can disallow that if you like

On to the next challenge!

Next challenge (1/3)

- This time, your boss wants you to put the company logo on every screen saver.

Screen Saver Company



Next challenge (2/3)

- This time, your boss wants you to put the company logo on every screen saver.

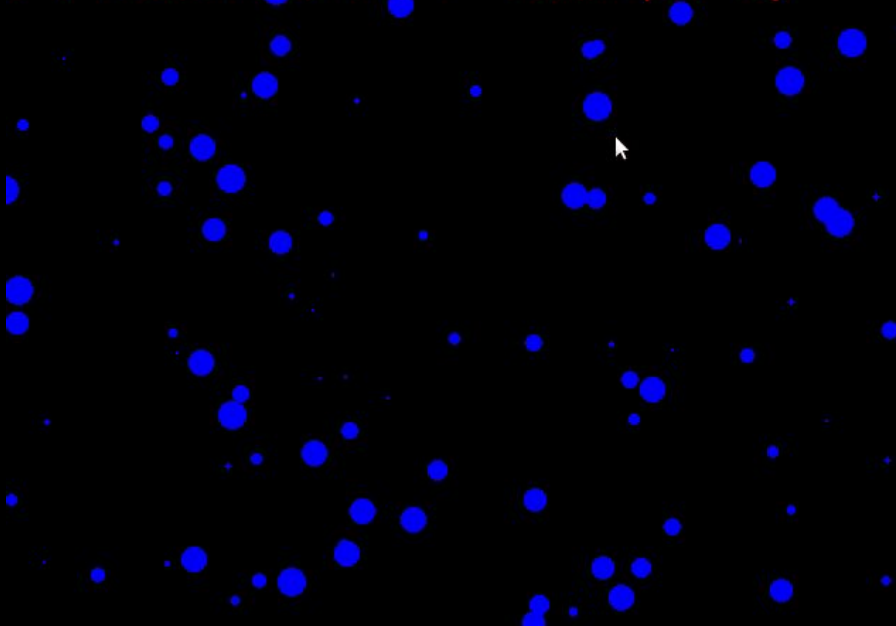


ScreenSaver Company

Next challenge (3/3)

- This time, your boss wants you to put the company logo on every screen saver.
- Your colleague is working on the actual legacy text rendering interface
 - text rendering involves:
 - loading font files, a custom string class, a drawable object class, and a deep hierarchy...
 - Oh, by the way, the text must always be green and uppercase letters.

Screen Saver Company



Problem we are solving (1/3)

- We have some some text that must always be uniform
 - This is based off of a legacy or third-party 'Text' class
- There are potentially a lot of steps needed to “fit” our specifications.

```
36 // Create some text
37 sf::Text text;
38 text.setFont(font);
39 text.setFillColor(sf::Color::Red);
40 // ... etc.
41 text.setString("ScreenSaver Company");
```

Problem we are solving (2/3)

- We have some some text that must always be uniform
 - This is based off of a legacy or third-party 'Text' class
- There are potentially a lot of steps needed to “fit” our specifications.
- One possible solution is to write a function we can pass all objects through
 - But what if we also need to modify other behaviors?
 - e.g. The `sf::Text.draw()` function?

```
36 // Create some text
37 sf::Text text;
38 text.setFont(font);
39 text.setFillColor(sf::Color::Red);
40 // ... etc.
41 text.setString("ScreenSaver Company");
```

```
20 void adaptText(sf::Text& original){
21 // Step 1:
22     original.setFillColor(sf::Color::Green);
23 // Step 2:
24 // ...
25 return;
26 }
```

Problem we are solving (3/3)

- We have some some text that must always be uniform

- This is based off of a legacy or third-party 'Text' class

- There are po

- One possible

function we c

- But what if we also need to modify other behaviors?
 - e.g. The `sf::Text.draw()` function?

```
36 // Create some text
37 sf::Text text;
38 text.setFont(font);
39 text.setFillColor(sf::Color::Red);
40 // ... etc. //
41 text.setString("ScreenSaver Company");
```

Introducing--Structural Design Patterns (Our second category)

```
21 // Step 1:
22 original.setFillColor(sf::Color::Green);
23 // Step 2:
24 // ...
25 return;
26 }
```

Structural Design Pattern

Focus on using inheritance to compose interfaces and create new objects

Structural Design Patterns

- *Usually* focus on using inheritance to compose interfaces and create new objects
 - This gives us the properties of the 'previous' object for which we can implement as needed.
- I'm now going to show the adapter pattern
 - (Which can be implemented in several ways)



Adapter Pattern

Converts the interface of a class into another interface the client expects

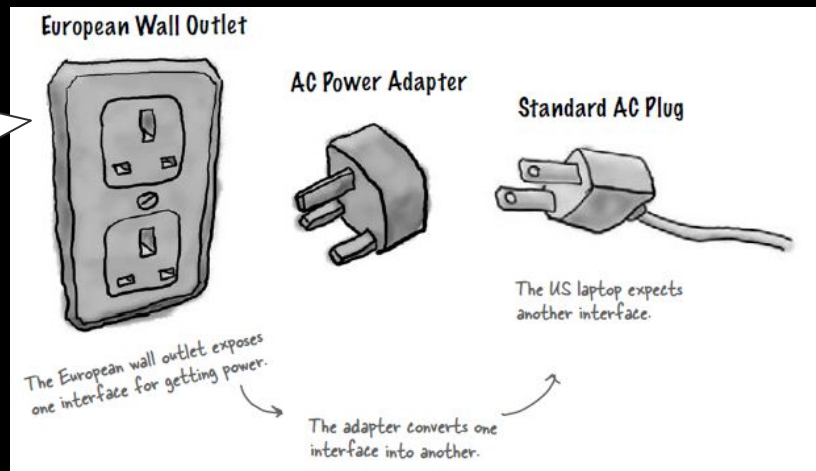
Adapter Pattern (1/2)

- **Pattern:** Converts the interface of a class into another interface the client expects
- **Common Uses:**
 - Writing a 'wrapper' to update or otherwise make an interface work
 - (i.e. updating a legacy codebase with some new functionality)
 - (e.g. I have often used, making various 'string' classes work together)

Adapter Pattern (2/2)

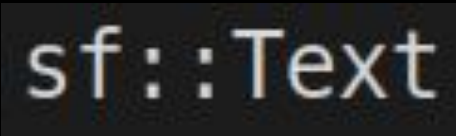
- **Pattern:** Converts the interface of a class into another interface the client expects
- **Common Uses:**
 - Writing a 'wrapper' to update or otherwise make an interface work
 - (i.e. updating a legacy codebase with some new functionality)
 - (e.g. I have often used, making various 'string' classes work together)

- The 'AC Power Adapter' is what we are going to build, so the client can use their Standard AC Plug in the correct interface
 - (Image from Head First Design Patterns Chapter 7)



Problem we are solving

- Problem:
 - Sometimes we have a class that does not quite have the functionality we want.
 - i.e. the `sf::Text` class
 - We want a simple way to update our software, without potentially breaking old functionality



```
sf::Text
```

A First Take at the Adapter Pattern (1/3)

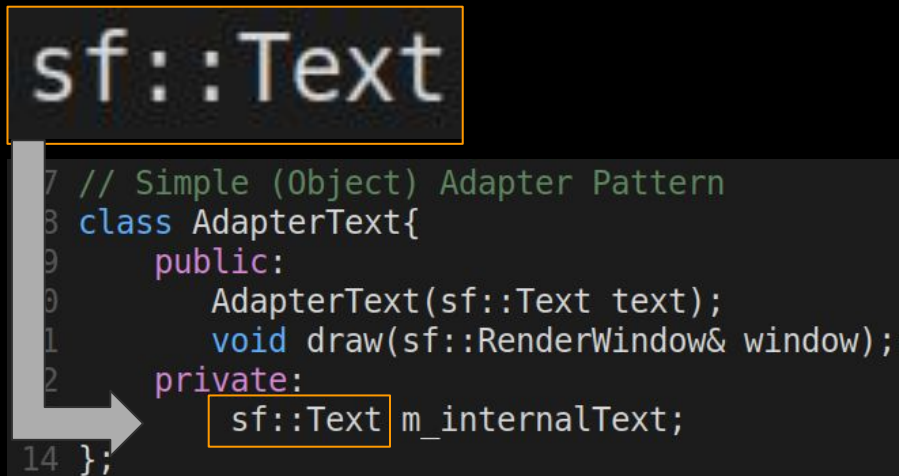
- Here is a new class
 - It stores an instance of the old type(that we want to *adapt*) as a member variable
- Our constructor of our new 'AdapterText' class takes in this type as a parameter

sf::Text

```
7 // Simple (Object) Adapter Pattern
8 class AdapterText{
9     public:
10         AdapterText(sf::Text text);
11         void draw(sf::RenderWindow& window);
12     private:
13         sf::Text m_internalText;
14 };
```

A First Take at the Adapter Pattern (2/3)

- Here is a new class
 - It stores an instance of the old type(that we want to *adapt*) as a member variable
- Our constructor of our new 'AdapterText' class takes in this type as a parameter
 - Note: We could inherit sf::Text as a base class, but let's make some assumptions for pedagogical purposes:
 - Assume there is a deep hierarchy, so we'd have to implement many member functions



```
7 // Simple (Object) Adapter Pattern
8 class AdapterText{
9     public:
10         AdapterText(sf::Text text);
11         void draw(sf::RenderWindow& window);
12     private:
13         sf::Text m_internalText;
14 };
```

A First Take at the Adapter Pattern (3/3)

- Here is an implementation in the constructor as well as a new behavior to draw our text
 - Note: We can add more *private* member functions to help perform the work done in the constructor.

```
7 // Simple (Object) Adapter Pattern
8 class AdapterText{
9     public:
10         AdapterText(sf::Text text);
11         void draw(sf::RenderWindow& window);
12     private:
13         sf::Text m_internalText;
14 };
```

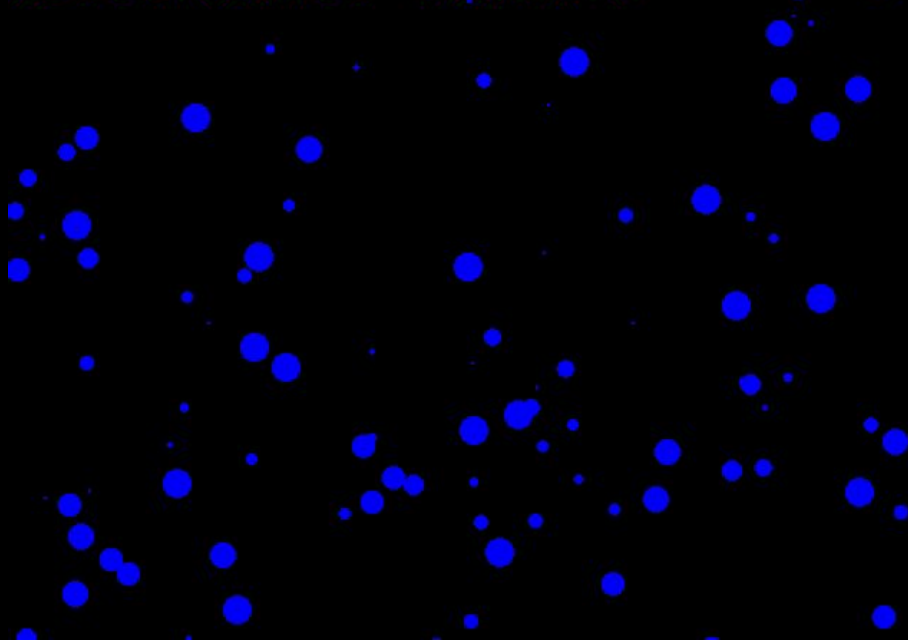
```
6 AdapterText::AdapterText(sf::Text text){
7     m_internalText = text;
8
9     m_internalText.setCharacterSize(24);
10    m_internalText.setFillColor(sf::Color::Green);
11    std::string original = text.getString();
12    for(size_t i=0; i < original.length();i++){
13        original[i] = std::toupper(original[i]);
14    }
15    m_internalText.setString(original);
16
17 }
18
19 void AdapterText::draw(sf::RenderWindow& window){
20     window.draw(m_internalText);
21 }
```

Adapter Pattern Usage

```
43 // Create an Adapter sometime after  
44 // our previous Text was created.  
45 AdapterText adapterText(text);
```

- Now we can do a quick ‘wrapper’ of our old `sf::Text` objects to make them render appropriately
 - Text is now always uppercase
 - Text is now always green
 - And we can adapt further functionality as need.

SCREENSAVER COMPANY



Adapter Pattern: Pros and Cons

- Pros

- Allows classes with otherwise incompatible interfaces to work together
 - (i.e. You can make things work after they were designed)
- Can work very quickly

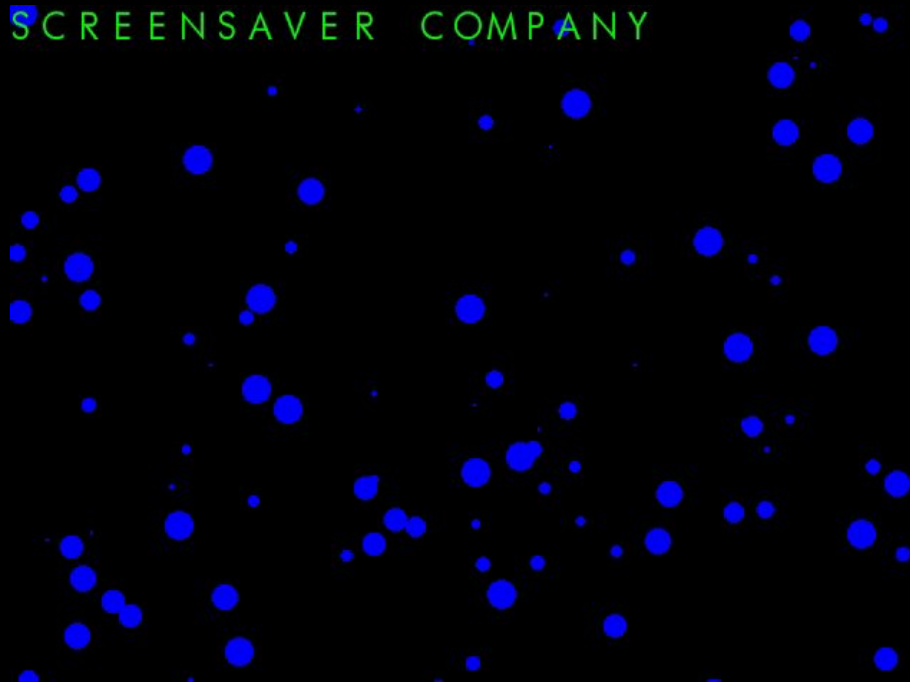
- Cons

- It may introduce more complexity into your code
 - If you own all of the code, it may be worthwhile to update the class to do something new.
 - Make sure you are solving the right problem!
- In my particular example, making extra copies to 'wrap' an object could be expensive!
 - (In which case, consider inheritance version of pattern or passing by reference to constructor)

Next Task

- Now your boss wants to fine tune this screensaver a bit
- They think it can be improved by adding some more circles!

SCREENSAVER COMPANY



A first refactoring (1/2)

- So you search the code and start finding for-loops that look like this:
 - You're pushing a fixed-amount of elements into a collection

```
48 // Create a circle shape
49 std::vector<Circle> circles;
50 for(int i=0; i < 100; i++){
51     Circle temp;
52     circles.push_back(temp);
53 }
```


A first refactoring (2/2)

- So you search the code and start finding for-loops that look like this:
 - You're pushing a fixed-amount of elements into a collection
- Later on you see some corresponding code to loop through all of the other elements

```
48 // Create a circle shape
49 std::vector<Circle> circles;
50 for(int i=0; i < 100; i++){
51     Circle temp;
52     circles.push_back(temp);
53 }
```

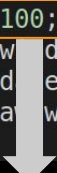
```
75 for(int i=0; i < 100; i++){
76     sf::Vector2u windowSize = window.getSize();
77     circles[i].update(windowSize.x,windowSize.y);
78     circles[i].draw(window);
79 }
```

A first refactoring - quick fix (1/2)

- So you search the code and start finding for-loops that look like this:
 - You're pushing a fixed-amount of elements into a collection
- Later on you see some corresponding code to loop through all of the other elements
 - This quick fix allows you to now loop-through your collection size without any worries
 - But...we can do better!

```
48 // Create a circle shape
49 std::vector<Circle> circles;
50 for(int i=0; i < 100; i++){
51     Circle temp;
52     circles.push_back(temp);
53 }
```

```
75 for(int i=0; i < 100; i++){
76     sf::Vector2u windowSize = window.getSize();
77     circles[i].update(windowSize.x,windowSize.y);
78     circles[i].draw(window);
79 }
```



```
75 for(int i=0; i < circles.size(); i++){
76     sf::Vector2u windowSize = window.getSize();
77     circles[i].update(windowSize.x,windowSize.y);
78     circles[i].draw(window);
79 }
```

A first refactoring - quick fix (2/2)

- So you search the code and start finding for-loops that look like this:

- You're pushing a fixed-amount of elements into a collection

```
48 // Create a circle shape
49 std::vector<Circle> circles;
50 for(int i=0; i < 100; i++){
51     Circle temp;
52     circles.push_back(temp);
```

- Later on you find a corresponding loop through all of

Introducing--Behavioral Design Patterns (Our third category)

- This quick loop-through your collection size without any worries
- But...we might be able to do better!

```
window.getSize();
size.x>windowSize.y);
```

```
75 for(int i=0; i < circles.size(); i++){
76     sf::Vector2u windowSize = window.getSize();
77     circles[i].update(windowSize.x>windowSize.y);
78     circles[i].draw(window);
79 }
```

Behavior Design Pattern

Focus on communication between objects

Behavior Design Patterns

- Focus on communication between objects
- Patterns are concerned with algorithms and responsibilities between objects.
 - The relationship I want to work with in our example, is how each object in a collection is linked, and how to traverse them.

Iterator Pattern

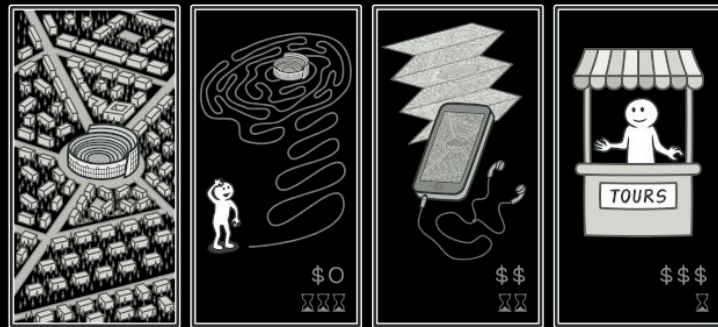
Provide sequential access to elements in a collection without exposing the representation

Iterator Pattern

- Problem:
 - Your data structures hold data in some type of collection (linked list, hashtable, array, etc.), and at some point a user may want to look at **all** of the data.
- Solution
 - We can encapsulate the iteration to ensure we have a consistent way to look through our data.
 - Perhaps from the first element to the last element
 - Or reversed to search from the last element to the first element
 - Or perhaps some other traversal (e.g. bread-first traversal or depth-first traversal, etc.)

Iterator Pattern Analogy

- How many possible traversals are there of Rome
 - source: <https://refactoring.guru/design-patterns/iterator>
- (i.e. there are many!)



A First Take at the Iterator Pattern (1/2)

- The first good news is that we get iterators for free with the STL
 - i.e. I did not implement anything
- We have also enforced the order of our traversal as well
 - It doesn't look like it, but we have made some improvements!
 - (Next slide)

```
75     for(int i=0; i < circles.size(); i++){  
76         sf::Vector2u windowSize = window.getSize();  
77         circles[i].update(windowSize.x,windowSize.y);  
78         circles[i].draw(window);  
79     }
```



```
75     for(std::vector<Circle>::iterator it = circles.begin();  
76         it != circles.end();  
77         it++)  
78     {  
79         sf::Vector2u windowSize = window.getSize();  
80         it->update(windowSize.x,windowSize.y);  
81         it->draw(window);  
82     }
```

A First Take at the Iterator Pattern (2/2)

- Each time we iterate, we are very explicitly doing so from the *beginning*
 - As opposed to *int i = 0*;
 - This is very clear in the code
- We are also maintaining that we iterate until the end
 - Equivalent to `circles.size()`
 - But more clear, and less likely to have 'off by one' error
- Finally, we are clearing moving to the next element in our iterator (`it++`)
 - We don't care what it means to move to the next element
 - But we will get is the next sequential element

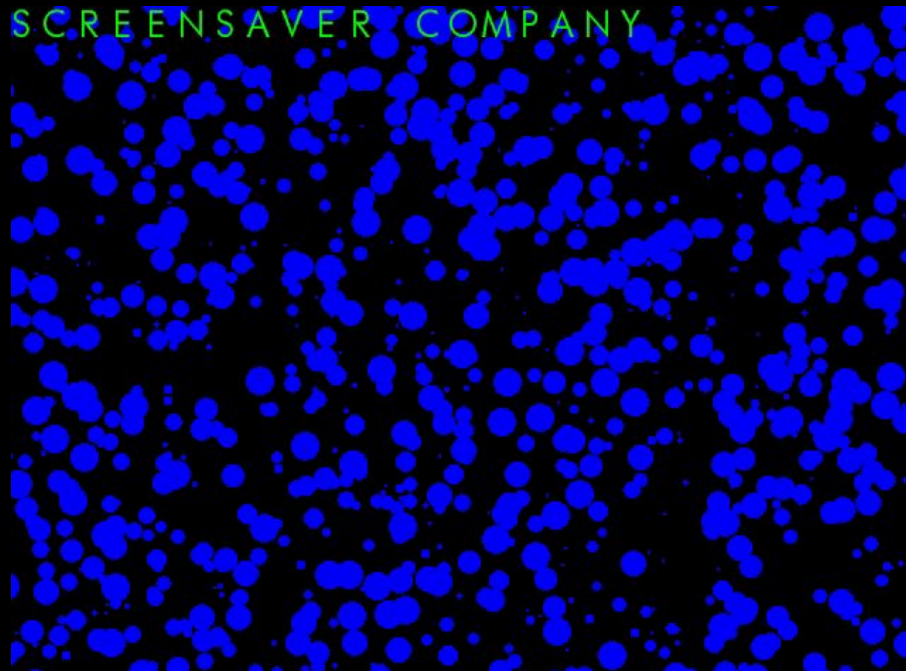
```
75     for(int i=0; i < circles.size(); i++){
76         sf::Vector2u windowSize = window.getSize();
77         circles[i].update(windowSize.x,windowSize.y);
78         circles[i].draw(window);
79     }
```

```
75     for(std::vector<Circle>::iterator it = circles.begin();
76         it != circles.end();
77         it++)
78     {
79         sf::Vector2u windowSize = window.getSize();
80         it->update(windowSize.x,windowSize.y);
81         it->draw(window);
82     }
```

Result

```
75     for(std::vector<Circle>::iterator it = circles.begin();  
76         it != circles.end();  
77         it++)  
78     {  
79         sf::Vector2u windowSize = window.getSize();  
80         it->update(windowSize.x,windowSize.y);  
81         it->draw(window);  
82     }
```

- The same traversal--but this time rendering 1000 circles
 - And we have also made our code more maintainable
 - i.e. We can change the underlying data structure
 - Maybe later we'll add collision detection and want to iterate through nodes only within a certain boundary, etc.



Iterators <http://www.cplusplus.com/reference/iterator/>

- So now, you should consider using the iterator pattern with at least the standard library collections
- Because the good news is, iterators are implemented for you in the STL!

<iterator>

Iterator definitions

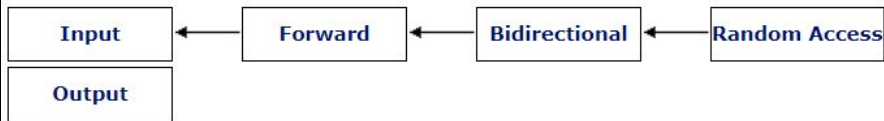
An *iterator* is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).

The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other kinds of iterators are possible. For example, each container type (such as a `list`) has a specific *iterator* type designed to iterate through its elements.

Notice that while a pointer is a form of iterator, not all iterators have the same functionality of pointers; Depending on the properties supported by iterators, they are classified into five different categories:

Iterator categories

Iterators are classified into five categories depending on the functionality they implement:



Input and output iterators are the most limited types of iterators: they can perform sequential single-pass input or output operations.

Forward iterators have all the functionality of input iterators and -if they are not *constant iterators*- also the functionality of output iterators, although they are limited to one direction in which to iterate through a range (forward). All standard containers support at least forward iterator types.

Bidirectional iterators are like forward iterators but can also be iterated through backwards.

Iterator Pros and Cons

- Pros
 - There is a single responsibility, it's very clear what the iterator should do
 - (i.e. Single Responsibility Principle)
 - A robust iterator may also protect you from insertions/removals in collections (especially in multithreaded programs)
- Cons
 - Applying the pattern may cost performance when used on simple types or data structures
 - You will have to profile and measure the performance

Summary

- We have looked at a small project where we can applications of three design patterns
 - Singleton
 - Adapter
 - Iterator
- Design patterns are not a magic solution, but may be a good start to solving your problem!
 - My recommendation is to continue learning more patterns and apply them to small projects to test out their utility

Going Further

Some things that may be useful for learning more design patterns

Unified Modeling Language (UML)

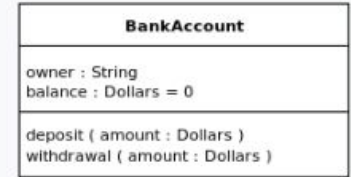
A compact way to represent software systems

What is the Unified Modeling Language (UML)?

- A general-purpose modeling language for software engineers to visualize the design of a system.
 - https://en.wikipedia.org/wiki/Unified_Modeling_Language
- Developed around 1994 through 1996.
- Useful for reading other folks work, useful as a software architect for larger systems
 - i.e. Drawing a diagram and planning before diving into code is wise!



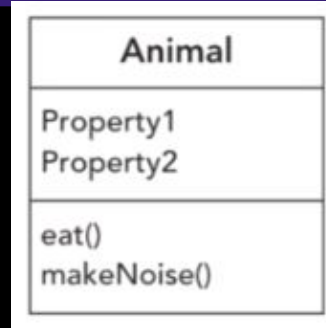
Component diagram



Class diagram

Sample UML for a Class Diagram (1/4)

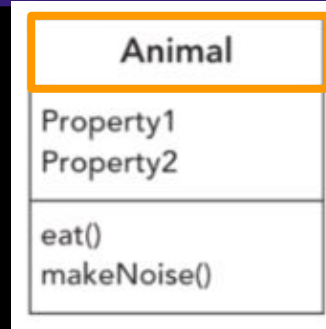
- Example of a class diagram
 - Class name
 - member variables (data)
 - member functions (behaviors)



```
1 class Animal{
2     private:
3         int property1;
4         int property2;
5
6     public:
7         void eat();
8         void makeNoise();
9 };
```

Sample UML for a Class Diagram (2/4)

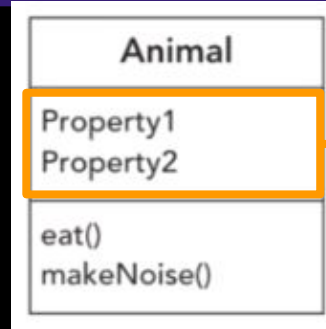
- Example of a class diagram
 - **Class name**
 - member variables (data)
 - member functions (behaviors)



```
1 class Animal{
2     private:
3         int property1;
4         int property2;
5
6     public:
7         void eat();
8         void makeNoise();
9 };
```

Sample UML for a Class Diagram (3/4)

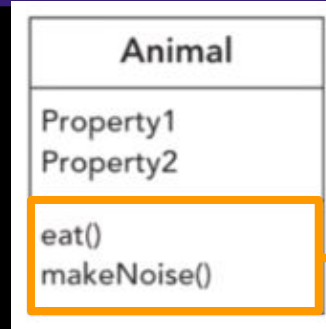
- Example of a class diagram
 - **Class name**
 - **member variables (data)**
 - **member functions (behaviors)**



```
1 class Animal{
2     private:
3         int property1;
4         int property2;
5
6     public:
7         void eat();
8         void makeNoise();
9 };
```

Sample UML for a Class Diagram (4/4)

- Example of a class diagram
 - **Class name**
 - member variables (data)
 - **member functions (behaviors)**



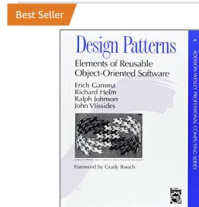
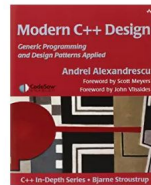
```
1 class Animal{
2     private:
3         int property1;
4         int property2;
5
6     public:
7         void eat();
8         void makeNoise();
9 };
```

The code defines the 'Animal' class. It has two private attributes, 'property1' and 'property2', and two public methods, 'eat()' and 'makeNoise()'. The methods are highlighted with an orange box, and an arrow points from this box to the corresponding diagram in the adjacent block.

Further Recommended Resources

Note: The running example (because it's nice to see code!) was developed using the excellent free SFML library (<https://www.sfml-dev.org/>).

Thank you to the developers!



Thank you CPPCON 2020!

Design Patterns

Mike Shah, Ph.D.

[@MichaelShah](#) | mshah.io

September 17, 2020

60 minutes | Introductory to Intermediate Level Audience

Thank you!