Classes

Lecture 6

Context

- In the previous lecture we talked about structures as tool to group *variables* into a single useful type

- Classes build on this idea (*encapsulation*), but with additional functionality*
  - Functions
  - Control over variable *visibility*

Terminology

- Like a `struct`, a `class` defines a data type
- A variable whose type is a class is called an *object* (sometimes referred to as an *instance* of the class)
- We have worked a lot with the **`string`** class – each string variable is an object

```
string foo; // declaring the foo object,
            // an instance of the string class
```

Encapsulating Code

- Structures allowed us to group together data (in *member variables*)
- Classes have this ability, but also allow us to bundle code as *member functions*
- This encapsulation allows us to provide safe and useful functionality without others having to know how the class operates
  - `str.length()` vs. `strlen(str)` vs. ends in `'\0'`
  - `str.at(i)` vs. `str[i]`

Member Functions

Member functions are like any other function, except:

- They are called with a specific object
- They have built-in access to the member variables/functions of that object

```
string foo( "Howdy!" );
cout << foo.length();
```

Access Comparison

`func(int a, char b, …);`

- Global variables/functions
  - **`cout`**
  - **`sqrt`**
- Arguments
  - **`a`**
  - **`b`**
- Local variable(s)

`obj.func(int a, char b, …);`

- Global variables/functions
  - **`cout`**
  - **`sqrt`**
- Arguments
  - **`a`**
  - **`b`**
- Local variable(s)
- Member variables of **`obj`**
- Member functions of **`obj`**
- Pointer to **`obj`**
  - **`this`**

Example

```
struct MyDate
{
    string month;
    int day;
    int year;
};

void output(const MyDate& md)
{
    cout << md.month << " "
         << md.day << ", "
         << md.year << endl;
}


…

output( bday );
```

```
class MyDate
{
public:
    string month;
    int day;
    int year;

    void output()
    {
        cout << month << " "
             << day << ", "
             << year << endl;
    }
};

…

bday.output();
```

Example (also valid)

```
struct MyDate
{
    string month;
    int day;
    int year;
};

void output(const MyDate& md)
{
    cout << md.month << " "
         << md.day << ", "
         << md.year << endl;
}


…

output( bday );
```

```
class MyDate
{
public:
    string month;
    int day;
    int year;

    void output()
    {
        cout << this->month << " "
             << this->day << ", "
             << this->year << endl;
    }
};

…

bday.output();
```

When to Use `this`

- Required if an argument name conflicts with a member variable name

- Optional to be explicit/clear about which variable/function the code is accessing

Example

```
class MyDate
{

public:
    string month;
    int day;
    int year;

    void output()
    {
        cout << month << " "
             << day << ", "
             << year << endl;
    }

    void setYear(int year)
    {
        this->year = year;
    }

};
```

Making Your Own Member Functions

1. Declare the function
2. Define the function
3. Use the function

Declaring a Member Function

Same as declaring any other function, but must be done within the class definition

```
class MyDate
{
public:
    string month;
    int day;
    int year;

    void output();
    void setYear(int year);
    int getCentury();
};
```

Defining a Member Function

- Similar to regular functions, there are two options: define with declaration, or define separately
- There are good reasons to separate declaration from definition (we will cover some of these later)
- For this class you should always define separately, and remember to comment the declaration (as well as any inline comments you see fit in the definition)

Example

```
class MyDate
{
public:
    string month;
    int day;
    int year;

    void output()
    {
        cout << month << " "
             << day << ", "
             << year << endl;
    }
};
```

```
class MyDate
{
public:
    string month;
    int day;
    int year;

    void output();
};

...

void MyDate::output()
{
    cout << month << " "
         << day << ", "
         << year << endl;
}
```

Separate Member Function Definition

- When defining member functions, remember to preface the function name with the class name and *scope resolution operator* (`::`)

```
<return> <class>::<function>(<args>)
{
}
```

- If you forget, C++ will attempt to define the function without any connection to the class
  – May lead to errors if the function accessed member variables/functions
  – Likely to cause a linker error for *undefined symbol* when other code attempts to use the member function

Using Member Functions

- Once a member function has been declared and defined, it can be used like member variables via the dot (.) operator

```
MyDate bday;
bday.month = "March";
bday.day = 15;
bday.year = -44;
bday.output();
```

- When dealing with object pointers, you can dereference and use the dot operator, or arrow shortcut

```
MyDate *d_p = &bday;
(*d_p).output();
d_p->output();
```

Exercise

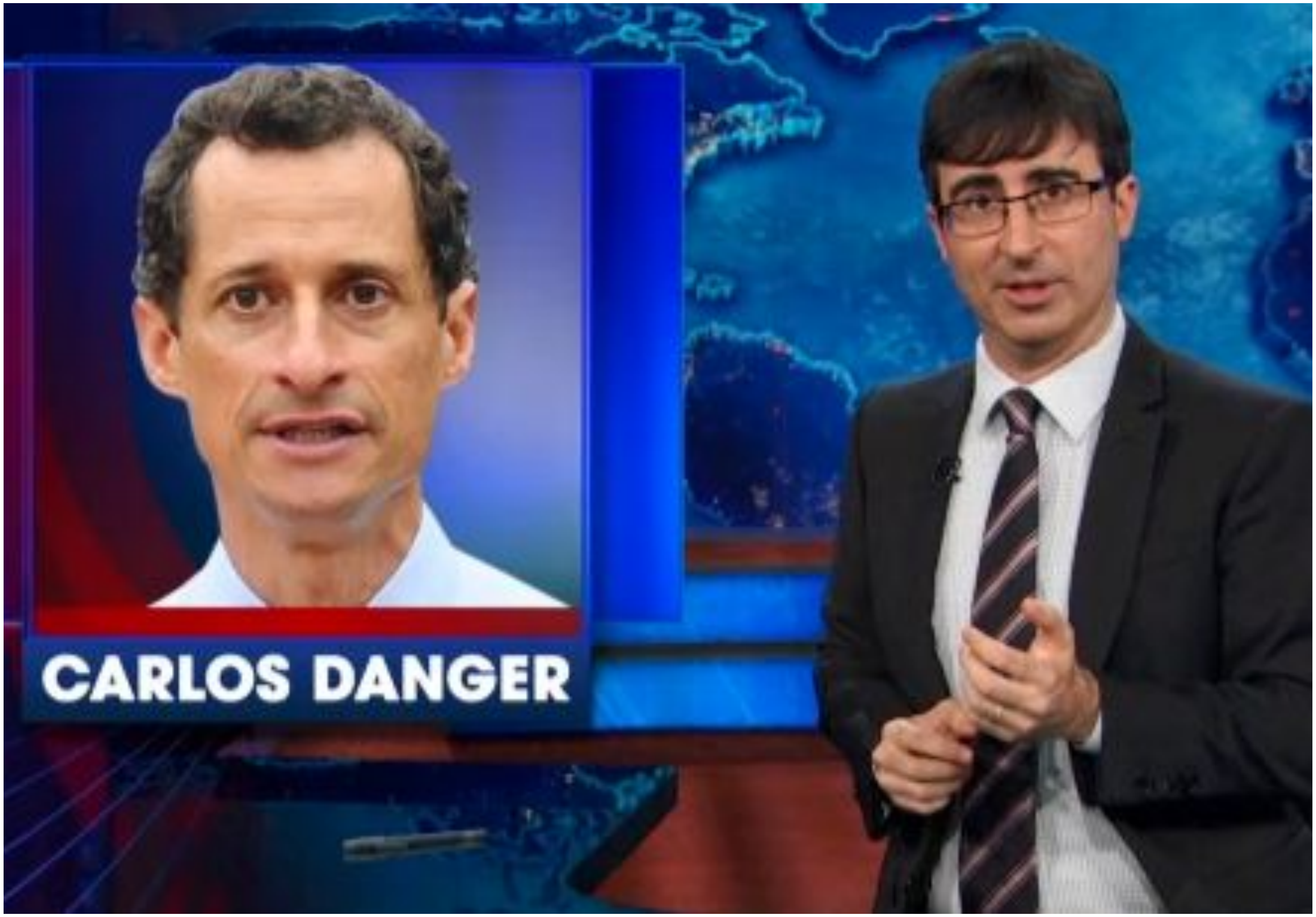
Create a class representing a circle. Add a single member variable, **`radius`**. Add two member functions to your circle class: **`output()`** should print the value of the member variable, **`area()`** should return the area of the circle. Write a **`main`** function to test the class.

Answer

```
#include <iostream>
using namespace std;

class Circle
{
public:
	void output();
	double area();

	double radius;
};

void Circle::output()
{
	cout << "Radius: " << radius;
}

double Circle::area()
{
	return ( 3.14159 * radius * radius );
}
```

```
int main()
{

	Circle c;

	c.radius = 2.0;
	c.output();
	cout << endl  << "Area: "
		<< c.area() << endl;

	return 0;
}
```

Danger!



Danger!

- In the previous example, there is nothing to prevent code from directly setting the radius member variable of the class to a negative value

- A central tenant of Object Oriented Programming (OOP) is *information hiding*
  - Protects client code from unnecessarily accessing aspects of the system, especially those that may change over time

Member Access Level

- To support information hiding, C++ classifies every class member (variables and functions) into a fixed access level
  - `public`: accessible by all code
  - `private`: accessible by member functions of the class (and a `friend`, discussed later)

- Later in the course we will discuss *inheritance*, which will involve a third access level (`protected`)

Why `private`?

- By making members `private`, you ensure they are not used outside of class member functions
  - This is typically done for all variables
- Functions are made `private` if they are only used internally in the class, and should not be called by a programmer that is utilizing the class
- In other words, `private` members are used to *hide* the implementation details of a class

Setting Member Access Level

- By default, all members of a class are **private**
- You change the level for one or more members by placing the keyword above them with a colon

```
class my_class
{
    int i; // private variable
public:
    void some_func(); // public function
    double x; // public variable
private:
    int fun(int c); // private function
    bool b; // private variable
public:
    double y; // public variable
};
```

Example

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
     int get();
     void set(int x);

private:
     int x;
};

void NaturalNumber::set(int x)
{
     if ( x >= 0 )
          this->x = x;
}

int NaturalNumber::get()
{
     return x;
}
```

```
int main()
{

     NaturalNumber num;

     num.set( 7 );
     cout << num.get() << endl;

     num.set( -1 );
     cout << num.get() << endl;

     num.set( 11 );
     cout << num.get() << endl;

     return 0;
}
```

7

7

11

Accessor and Mutator Functions

- You should almost always be making variables private in your classes
- However, to be useful, client code will need at least indirect access to some of these variables
- Functions that allow *read* access are called *accessor* functions, sometimes *getters*
- Functions that allow *write* access are called *mutator* functions, sometimes *setters*

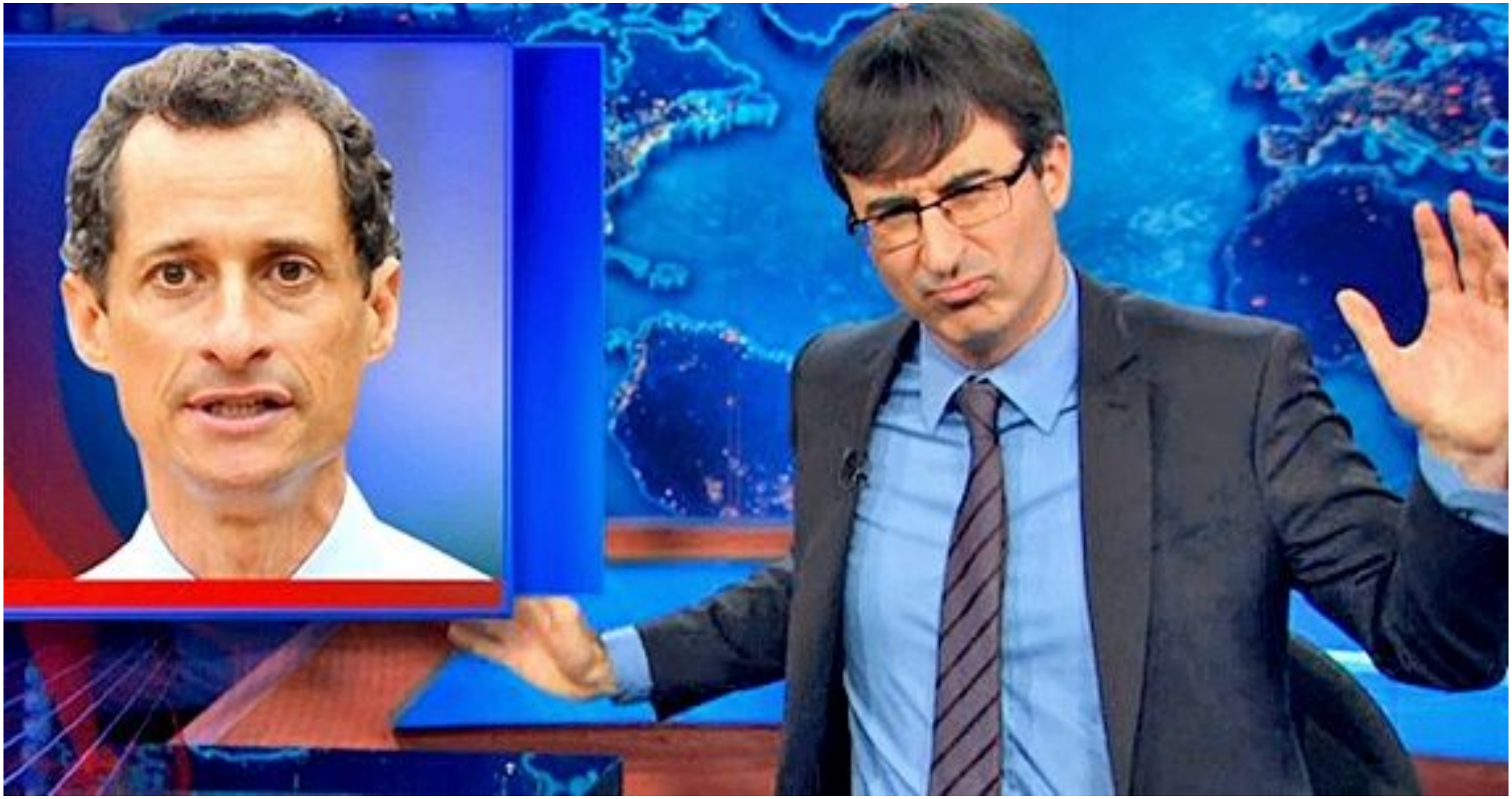
Fix the Circle!

Create a class representing a circle. Add a single member variable, **`radius`**. Add <u>three</u> member functions to your circle class: **`output()`** should print the value of the member variable, **`area()`** should return the area of the circle, and **setRadius()** should allow client code to set a valid radius (>0). Write a **`main`** function to test the class.

Answer

```cpp
#include <iostream>
using namespace std;

class Circle
{
public:
	void output();
	double area();
	void setRadius(double radius);

private:
	double radius;
};

void Circle::output()
{
	cout << "Radius: " << radius;
}

double Circle::area()
{
	return ( 3.14159 * radius * radius );
}

void Circle::setRadius(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
}

int main()
{
	Circle c;

	c.setRadius( 2.0 );
	c.output();
	cout << endl  << "Area: "
		<< c.area() << endl;

	c.setRadius( -7.0 );
	c.output();
	cout << endl  << "Area: "
		<< c.area() << endl;

	return 0;
}
```

More Danger!



More Danger!

In the previous example, there is nothing to stop client code from using an accessor function before the object is in a valid state

```
Circle c;
c.output(); // garbage!
cout << c.area(); // 3.14159 * garbage²
```

Constructors

- Constructors are special member functions that are used for *initialization*
- A class can have multiple constructors that have different argument lists, but each object can only be initialized with one constructor
- The function name for a constructor is the same as the name of the class, there is no return value
- Except under very special circumstances, constructors should always be `public`

Calling a Constructor

- A constructor is called automatically when you declare an object
  - Also for dynamic allocation on **`new`**
- No constructor can be called after an object is declared
- Only one constructor can be called per object, and one constructor is always called
- You specify the arguments in parentheses after the variable name

Example

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get();
	void set(int x);
	NaturalNumber(int x);

private:
	int x;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get()
{
	return x;
}
```

```
NaturalNumber::NaturalNumber(int x)
{
	if ( x >= 0 )
		this->x = x;
	else
		this->x = 0;
}

int main()
{
	NaturalNumber num( 3 );
	cout << num.get() << endl; // 3

	num.set( 7 );
	cout << num.get() << endl; // 7

	num.set( -1 );
	cout << num.get() << endl; // 7

	num.set( 11 );
	cout << num.get() << endl; // 11

	return 0;
}
```

Fix the Circle! (2)

Create a class representing a circle. Add a single member variable, **`radius`**. Add three member functions to your circle class: **`output()`** should print the value of the member variable, **`area()`** should return the area of the circle, and **setRadius()** should allow client code to set a valid radius (>0). Add a constructor that takes as an argument the initial radius – if it isn't valid, default to 1. Write a **`main`** function to test the class.

Answer

```
#include <iostream>
using namespace std;

class Circle
{
public:
      void output();
      double area();
      void setRadius(double radius);
      Circle(double radius);

private:
      double radius;
};

void Circle::output()
{
      cout << "Radius: " << radius;
}

double Circle::area()
{
      return ( 3.14159 * radius * radius );
}

void Circle::setRadius(double radius)
{
      if ( radius > 0 )
            this->radius = radius;
}

Circle::Circle(double radius)
{
      if ( radius > 0 )
            this->radius = radius;
      else
            this->radius = 1;
}

int main()
{
      Circle c( 3 );
      c.output(); // 3

      c.setRadius( 2.0 );
      c.output(); // 2

      c.setRadius( -7.0 );
      c.output(); // 2

      return 0;
}
```

Multiple Constructors

- You can define as many constructors as you want for each class, so long as they conform to the normal function overloading rules
- The argument lists have to be different, meaning different types or different numbers of arguments
- C++ automatically chooses the correct constructor based on the arguments provided

Default Constructor

- One special constructor is the *default* constructor
- This is the constructor used when no arguments are provided at object declaration
  - Example: **`string str;`**
- If you define no constructors for a class, the compiler automatically adds a default constructor that does nothing
- If you define *any* constructors for a class (not necessarily a default constructor), the compiler does NOT add a blank default constructor for you

Example

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get();
	void set(int x);
	NaturalNumber();
	NaturalNumber(int x);

private:
	int x;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get()
{
	return x;
}
```

```
NaturalNumber::NaturalNumber()
{
	x = 0;
}

NaturalNumber::NaturalNumber(int x)
{
	if ( x >= 0 )
		this->x = x;
	else
		this->x = 0;
}

int main()
{
	NaturalNumber num( 3 );
	cout << num.get() << endl; // 3

	num.set( 7 );
	cout << num.get() << endl; // 7

	NaturalNumber num2;
	cout << num2.get() << endl; // 0

	return 0;
}
```

Fix the Circle! (3)

Create a class representing a circle. Add a single member variable, **`radius`**. Add <u>three</u> member functions to your circle class: **`output()`** should print the value of the member variable, **`area()`** should return the area of the circle, and **setRadius()** should allow client code to set a valid radius (>0). Add a constructor that takes as an argument the initial radius – if it isn't valid, default to 1. Also add a default constructor that sets the radius to 1. Write a **`main`** function to test the class.

Answer

```
#include <iostream>
using namespace std;

class Circle
{
public:
	void output();
	double area();
	void setRadius(double radius);
	Circle();
	Circle(double radius);

private:
	double radius;
};

void Circle::output()
{
	cout << "Radius: " << radius;
}

double Circle::area()
{
	return ( 3.14159 * radius * radius );
}

Circle::Circle()
{
	radius = 1;
}
```

```
void Circle::setRadius(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
}

Circle::Circle(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
	else
		this->radius = 1;
}

int main()
{
	Circle c( 3 );
	c.output(); // 3

	c.setRadius( 2.0 );
	c.output(); // 2

	Circle c2;
	c2.output(); // 1

	return 0;
}
```

Destructors

- A destructor is an *optional* member function that is called when a variable goes out of scope
  - Also for dynamic allocation on **`delete`**

- The function name for a constructor is the same as the name of the class, prefaced by the tilde (**~**) symbol, there is no return value

- There can be up to one destructor, and it can take no arguments

- Used to clean up after the class
  - Especially useful to release any dynamically allocated memory

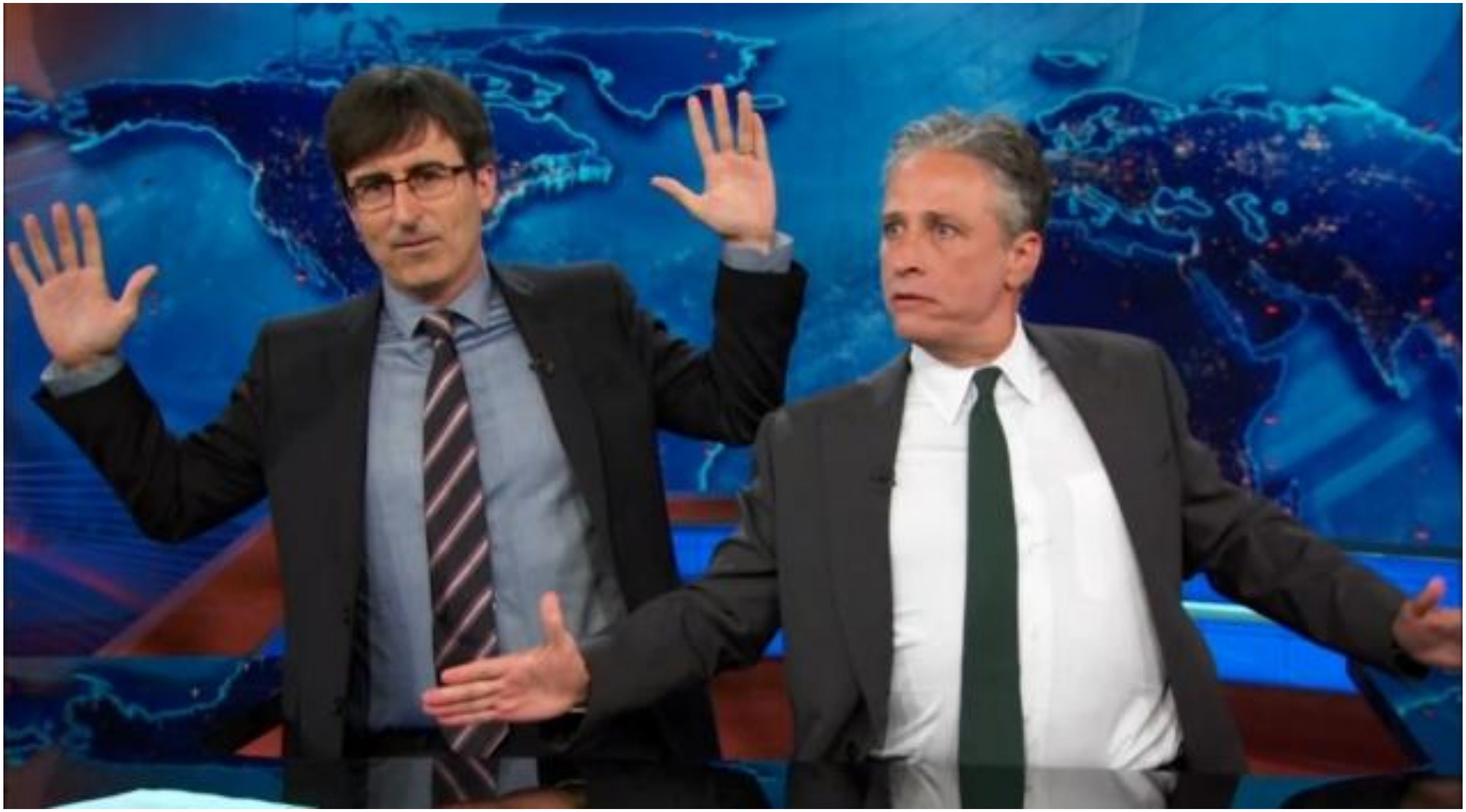
Example

```
#include <iostream>
using namespace std;

class MemoryHog
{
public:
	MemoryHog(int size);
	~MemoryHog();

private:
	int *array;
	int size;
};

MemoryHog::MemoryHog(int size)
{
	array = new int[ size ];
	this->size = size;
	cout << "Wasting " << size
	     << " ints!" << endl;
}

MemoryHog::~MemoryHog()
{
	delete[] array;
	cout << "Gave back " << size
	     << " ints!" << endl;
}
```

```
int main()
{
	MemoryHog hog1( 100 );
	MemoryHog* hog2;

	{
		MemoryHog hog3( 300 );
		hog2 = new MemoryHog( 200 );
	}

	delete hog2;

	return 0;
}
```

```
Wasting 100 ints!
Wasting 300 ints!
Wasting 200 ints!
Gave back 300 ints!
Gave back 200 ints!
Gave back 100 ints!
```

Even More Danger!



Even More Danger!

- In a previous lecture we learned how to pass classes/structures by reference, while protecting them from being changed

```
const type& object
```

- In order to adhere to this "contract" (i.e. will not change the object), C++ needs to know which member functions do not change member variables

Motivating Example (1)

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get();
	void set(int x);
	NaturalNumber();

private:
	int x;
	int gotten;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get()
{
	gotten++; // changes this
	return x;
}
```

```
NaturalNumber::NaturalNumber()
{
	x = 0;
	gotten = 0;
}

void outputNumber(NaturalNumber& num)
{
	cout << num.get() << endl;
}

int main()
{
	NaturalNumber num;
	outputNumber( num );
}
```

Motivating Example (2)

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get();
	void set(int x);
	NaturalNumber();

private:
	int x;
	int gotten;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get()
{
	gotten++; // changes this
	return x;
}
```

```
NaturalNumber::NaturalNumber()
{
	x = 0;
	gotten = 0;
}

void outputNumber(const NaturalNumber& num)
{
	cout << num.get() << endl;
}

int main()
{
	NaturalNumber num;
	outputNumber( num );
}
```

Compile Error

```
In function 'void outputNumber(const NaturalNumber&)':
error: passing 'const NaturalNumber' as 'this' argument of
'int NaturalNumber::get()' discards qualifiers [-
fpermissive]
  cout << num.get() << endl;
```

The `const` Modifier (take 3)

- Place the `const` modifier *after* the argument list in a member function declaration and definition in order to promise C++ that the function does not change *any* member variables
- The compiler will now raise errors if this promise is not kept, either directly or by calling other non-`const` member functions

Motivating Example (3)

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get() const;
	void set(int x);
	NaturalNumber();

private:
	int x;
	int gotten;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get() const
{
	gotten++; // changes this
	return x;
}
```

```
NaturalNumber::NaturalNumber()
{
	x = 0;
	gotten = 0;
}

void outputNumber(const NaturalNumber& num)
{
	cout << num.get() << endl;
}

int main()
{
	NaturalNumber num;
	outputNumber( num );
}
```

Compile Error

```
In member function 'int NaturalNumber::get() const':
error: increment of member 'NaturalNumber::gotten' in read-only object
  gotten++; // changes this
```

Motivating Example (4)

```
#include <iostream>
using namespace std;

class NaturalNumber
{
public:
	int get() const;
	void set(int x);
	NaturalNumber();

private:
	int x;
	int gotten;
};

void NaturalNumber::set(int x)
{
	if ( x >= 0 )
		this->x = x;
}

int NaturalNumber::get() const
{
	return x;
}
```

```
NaturalNumber::NaturalNumber()
{
	x = 0;
	gotten = 0;
}

void outputNumber(const NaturalNumber& num)
{
	cout << num.get() << endl;
}

int main()
{
	NaturalNumber num;
	outputNumber( num ); // 0
}
```

Fix the Circle! (4)

Create a class representing a circle. Add a single member variable, **`radius`**. Add three member functions to your circle class: **`output()`** should print the value of the member variable, **`area()`** should return the area of the circle, and **setRadius()** should allow client code to set a valid radius (>0). Add a constructor that takes as an argument the initial radius – if it isn't valid, default to 1. Also add a default constructor that sets the radius to 1. Write a **`main`** function to test the class. Make sure the class satisfies *const correctness*.

Answer

```
#include <iostream>
using namespace std;

class Circle
{
public:
	void output() const;
	double area() const;
	void setRadius(double radius);
	Circle();
	Circle(double radius);

private:
	double radius;
};

void Circle::output() const
{
	cout << "Radius: " << radius;
}

double Circle::area() const
{
	return ( 3.14159 * radius * radius );
}

Circle::Circle()
{
	radius = 1;
}

void Circle::setRadius(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
}

Circle::Circle(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
	else
		this->radius = 1;
}

void outputCircle(const Circle& c)
{
	c.output();
	cout << endl;
	cout << "Area: " << c.area() << endl;
}

int main()
{
	Circle c( 3 );
	outputCircle( c );
	return 0;
}
```

Structures Revisited

- In C++, a `struct` is actually a `class` with default access level of `public`
  - Technically, you can use structures for any situation in which you can use a class

- In C, a `struct` only has public member variables (like presented here)

- Thus, to avoid two keywords for a single concept, most programmers in C++ will use classes for OOP and only use structures if it is in the spirit of a C structure (i.e. a *record* of public fields)

Exercise

Create a class representing a sphere. Add a single member variable, **`radius`**. Add four member functions to your circle class: **`getRadius()`** should return the value of the member variable, **`setRadius()`** should allow client code to set a valid radius (>0), **`surfaceArea()`** should return the surface area of the sphere, and **volume()** should return the volume of the sphere. Add a constructor that takes as an argument the initial radius – if it isn't valid, default to 1. Also add a default constructor that sets the radius to 1. Write a **`main`** function to test the class. Make sure the class satisfies *const correctness*.

Surface area: $4\pi r^2$

Volume: $\frac{4}{3}\pi r^3$

Answer

```
#include <iostream>
using namespace std;

class Sphere
{
public:
	double getRadius() const;
	double surfaceArea() const;
	double volume() const;
	void setRadius(double radius);
	Sphere();
	Sphere(double radius);

private:
	double radius;
};

double Sphere::getRadius() const
{
	return radius;
}

double Sphere::surfaceArea() const
{
	return ( 4.0*3.14159*radius*radius );
}

double Sphere::volume() const
{
	return ( (4.0/3.0)*3.14159*radius*radius*radius );
}
```

```
void Sphere::setRadius(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
}

Sphere::Sphere()
{
	radius = 1;
}

Sphere::Sphere(double radius)
{
	if ( radius > 0 )
		this->radius = radius;
	else
		this->radius = 1;
}

void outputSphere(const Sphere& s)
{
	cout << "Radius: " << s.getRadius() << endl
	     << "Surface Area: " << s.surfaceArea() << endl
	     << "Volume: " << s.volume() << endl;
}

int main()
{
	Sphere s( 4 );
	outputSphere( s );
	return 0;
}
```

Wrap Up

- A `class` defines a complex data type
  - It *encapsulates* member variables and functions
  - It abstracts away implementation from interface via member access levels

- Constructors are member functions that execute automatically to initialize an object

- Destructors are member functions that execute automatically to clean up after an object

- Use of `const` correctness can keep object state safe while simultaneously achieving efficient passing to as function arguments