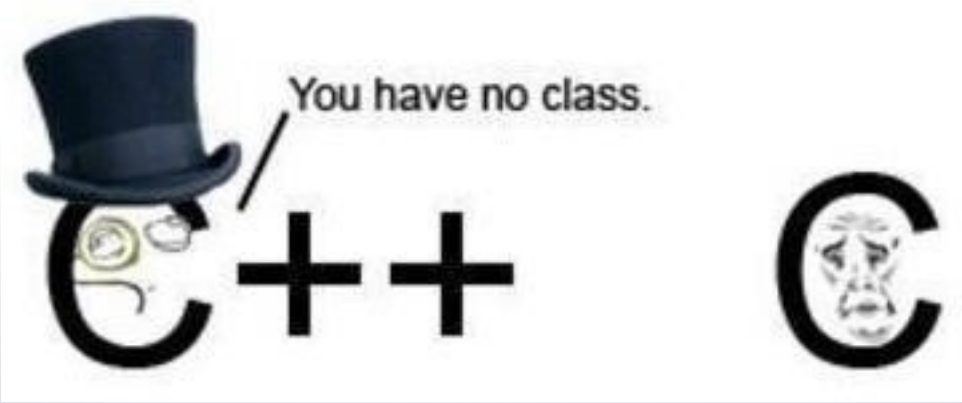
OOP

Classes, Objects, and ADTs

(oh my!)

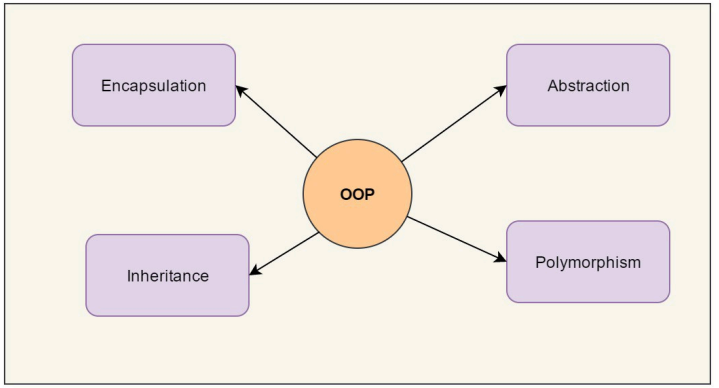
Week 2


You have no class.
C++
C

Object Oriented Programming

OOP

- OOP is a programming language model organized around objects rather than "actions" and data rather than logic.
- There are 4 major principles that make a language Object Oriented
  - Encapsulation
  - Data Abstraction
  - Polymorphism
  - Inheritance



Four Pillars of Object Oriented Programming

Encapsulation

- Hiding mechanism, hiding of data implementation
  - Private variables
  - Public accessor methods

Abstractions

- A concept or an Idea which is not associated with any particular instance
  - Expressing the “intent” and not the “implementation”
  - Abstract classes and virtual members

Polymorphism

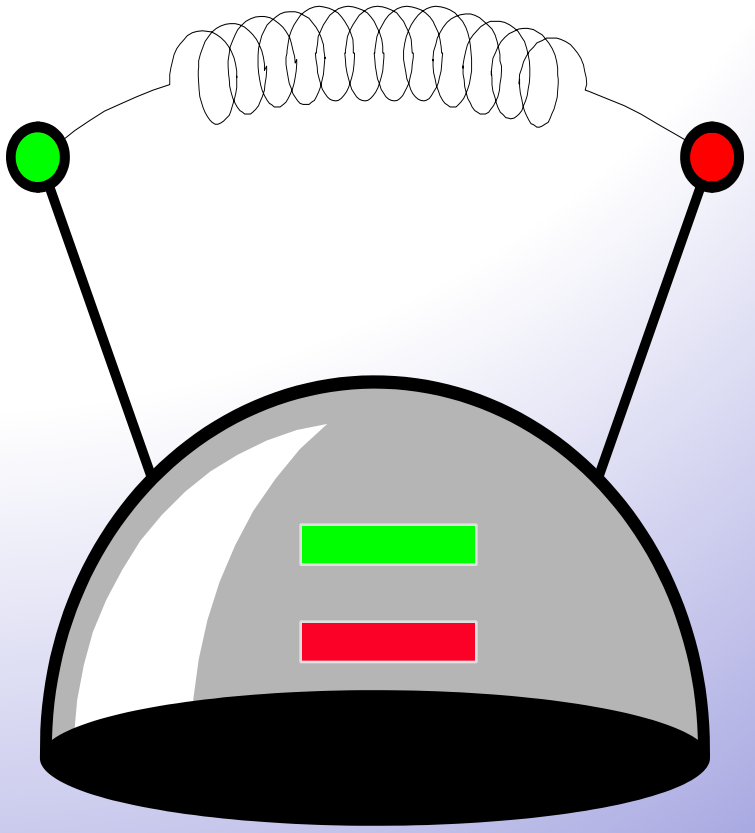
- One name, many forms!
  - What does a + b mean?

Inheritance

- “is a” or “has a” relationship between objects
  - Dog ”is a” mammal
  - Ferrari “is a” car, “has an” engine

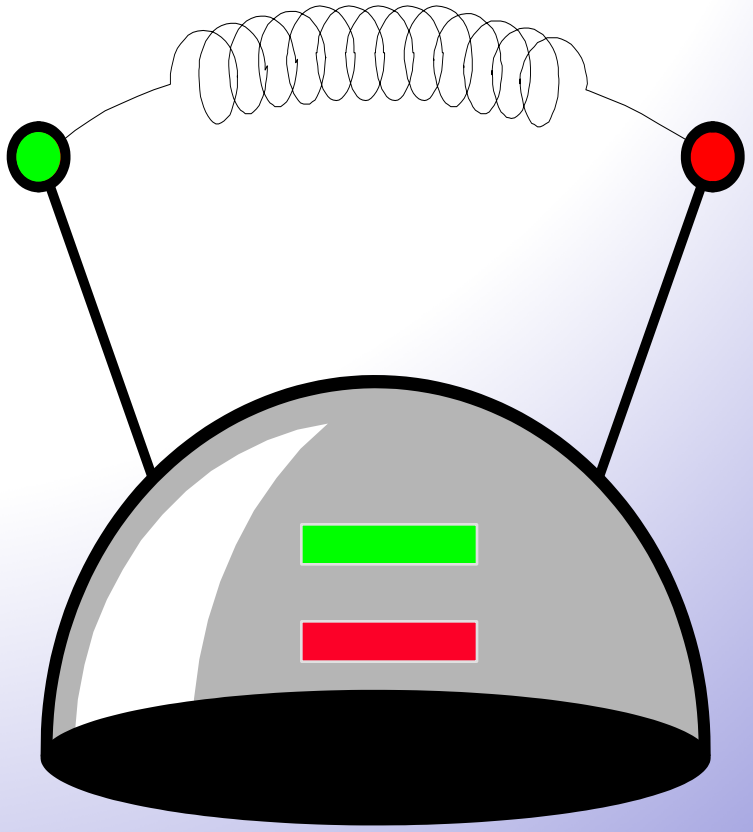
What is this “Object” ?

- There is no real answer to the question, but we’ll call it a “thinking cap”.
- The plan is to describe a thinking cap by telling you what actions can be done to it.

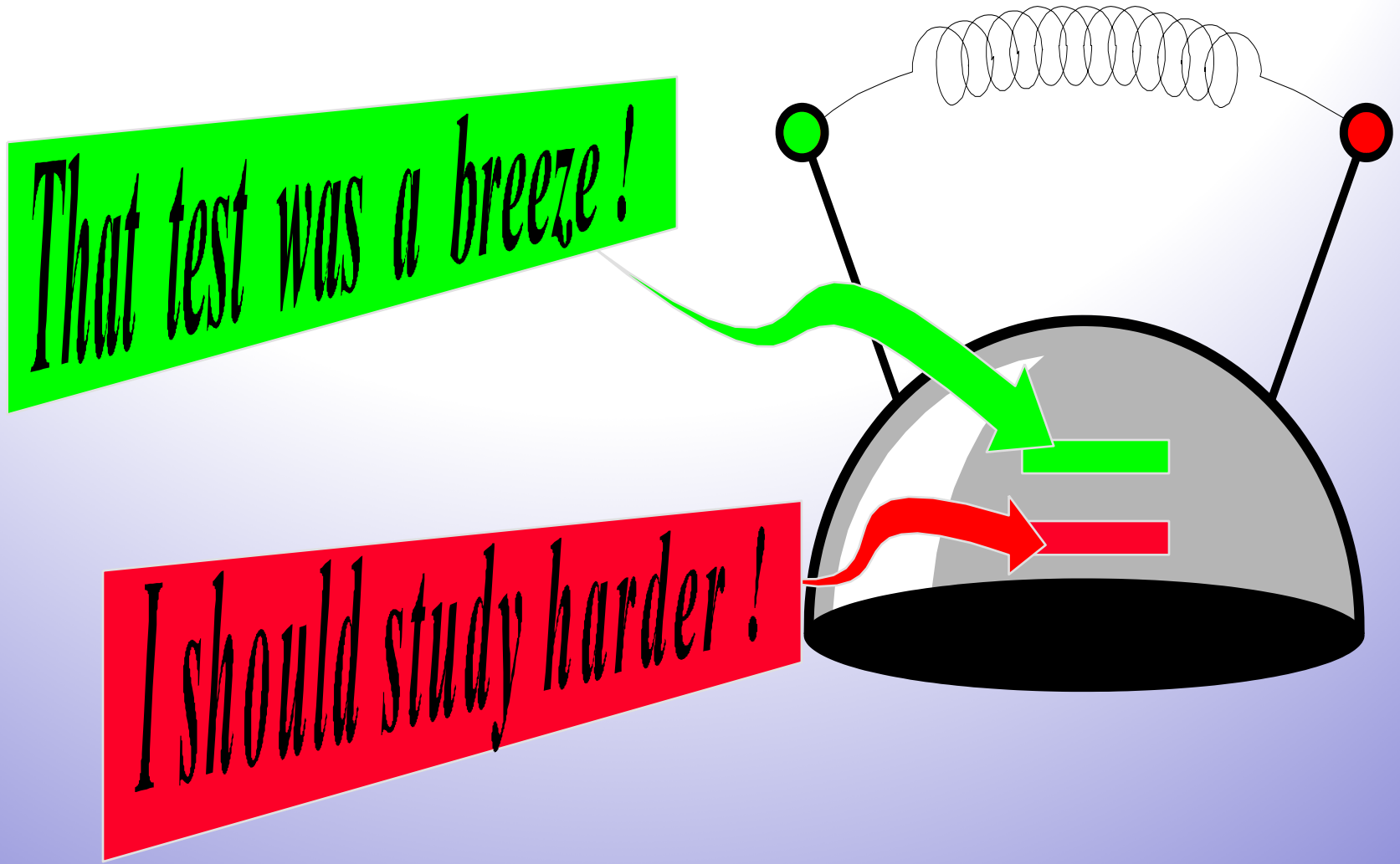


Using the Object's Slots

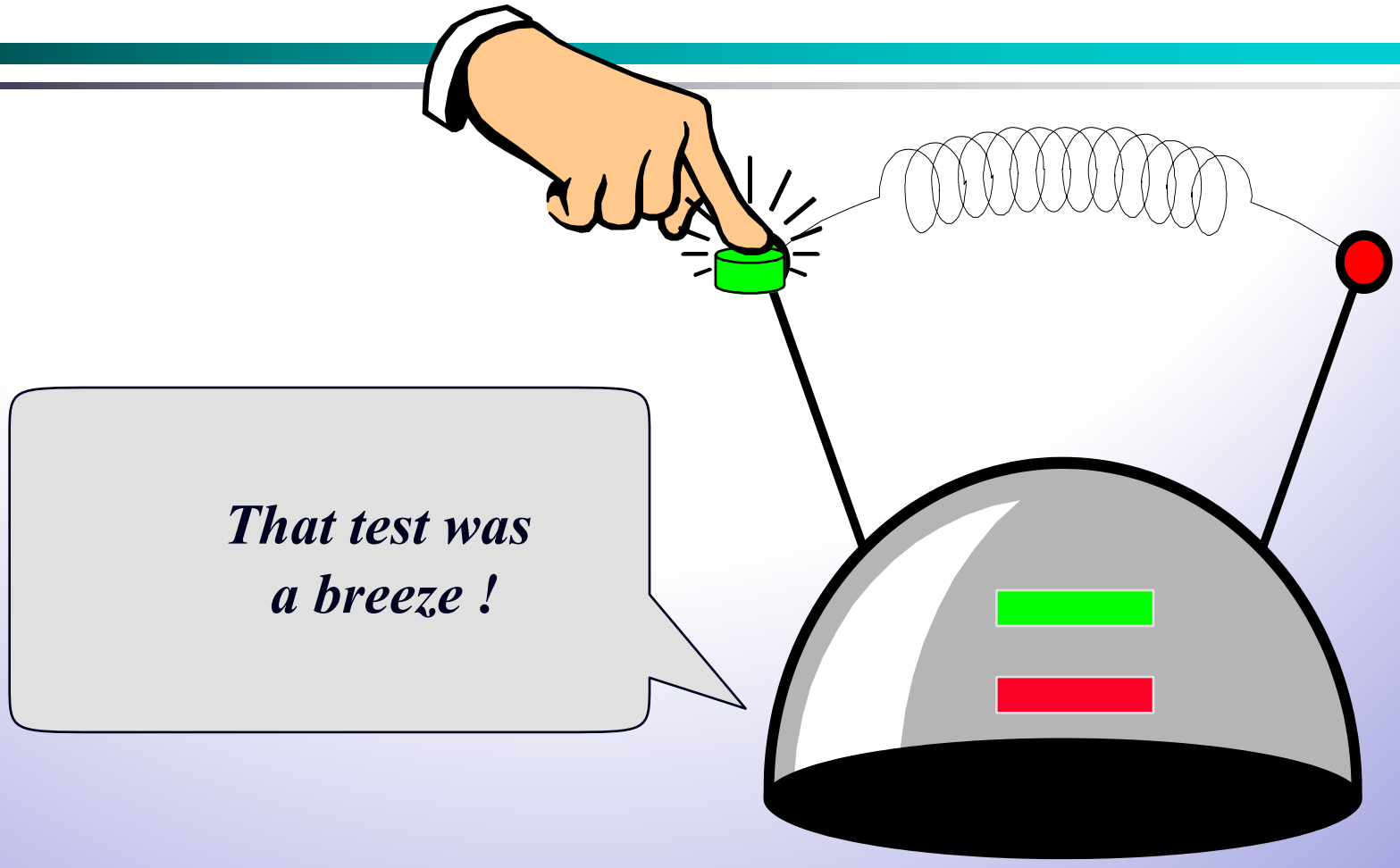
- You may put a piece of paper in each of the two slots (green and red), with a sentence written on each.
- You may push the green button and the thinking cap will speak the sentence from the green slot's paper.
- And same for the red button.



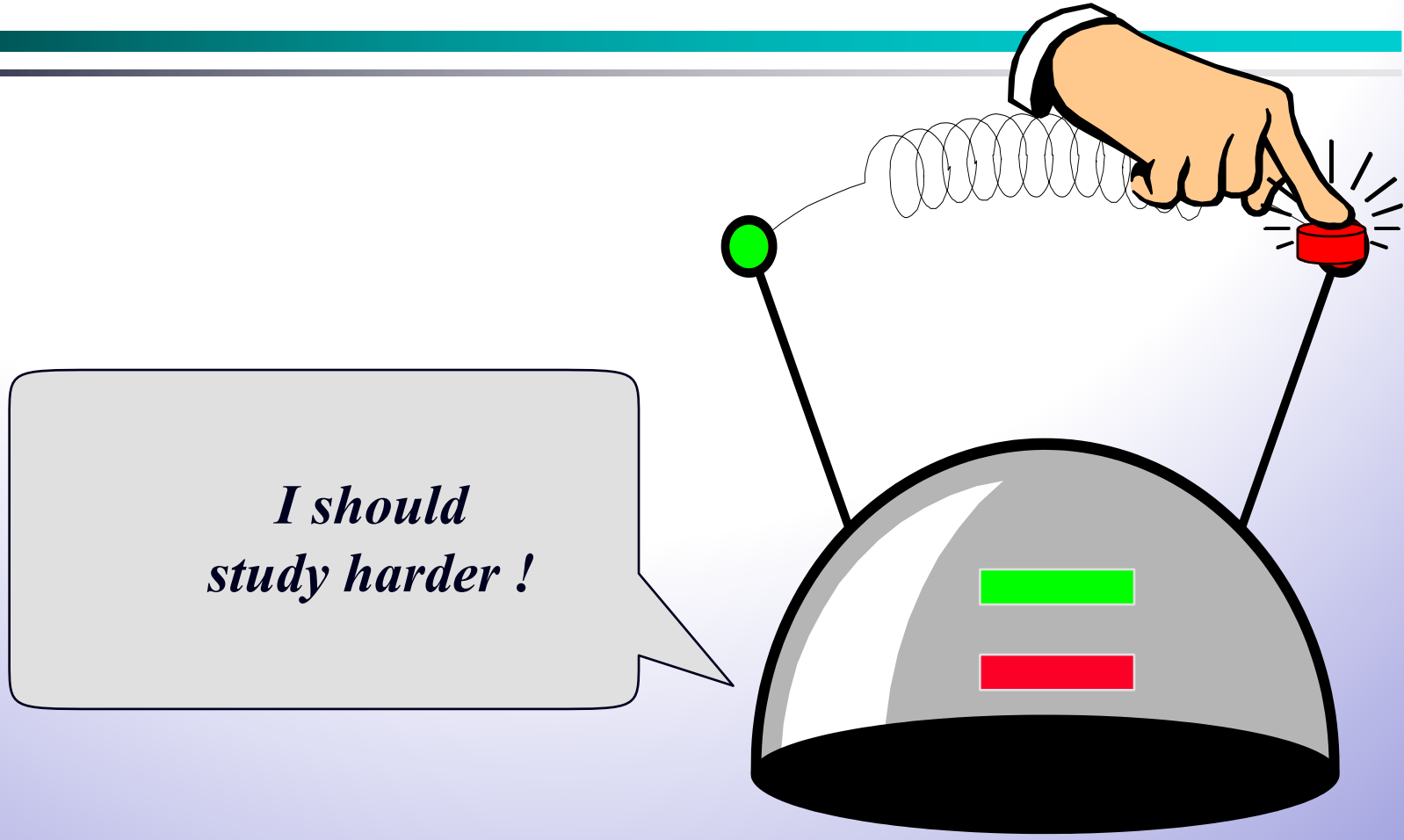
Example



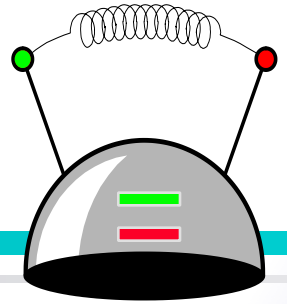
Example



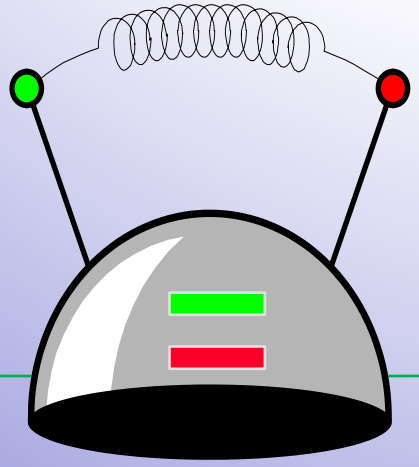
Example



Thinking Cap Implementation



- We can implement the thinking cap using a data type called a class.



```
class thinking_cap
{

        . . .

};
```

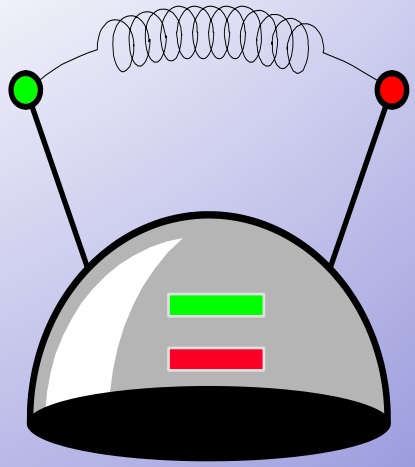
Class

A class is a new kind of data type

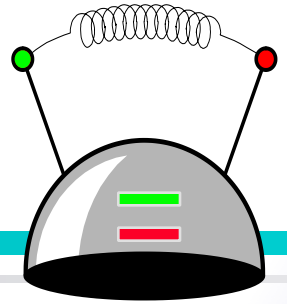
- A collection of data, such as integers, characters, and so on
- We implement data structures as classes
- A class has the ability to include special functions, called **member function** which are designed specifically to manipulate the class

Thinking_cap Class

```
class thinking_cap
{




};
```

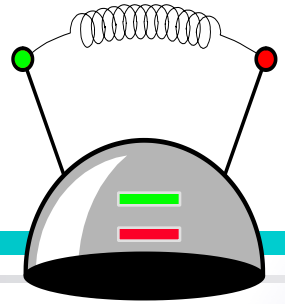


Thinking Cap Implementation



```
class thinking_cap
{
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( ) const;
    void push_red( ) const;
private:
    char green_string[50];
    char red_string[50];
};
```

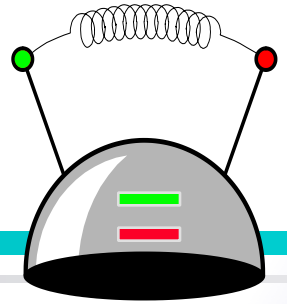
Thinking Cap Implementation



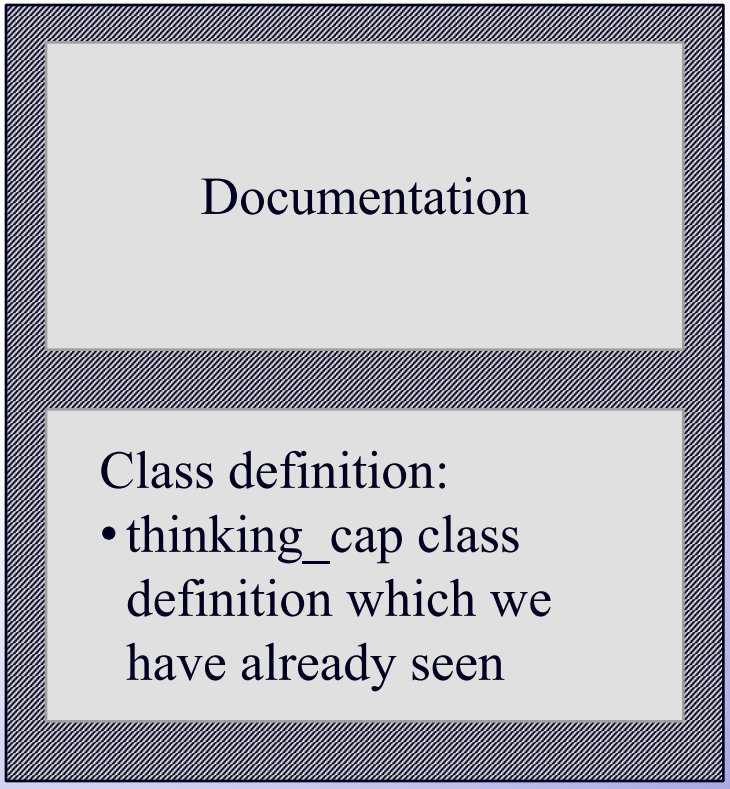
```
class thinking_cap
{
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( ) const;
    void push_red( ) const;
private:
    char green_string[50];
    char red_string[50];
};
```

This means that these functions will not change the data stored in a thinking_cap.

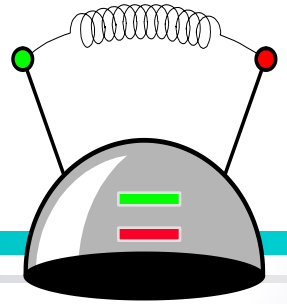
Files for the Thinking Cap



- The thinking_cap class definition, which we have just seen, is placed with documentation in a file called thinker.hxx, outlined here.
- The implementations of the three member functions will be placed in a separate file called thinker.cxx, which we will examine in a few minutes.



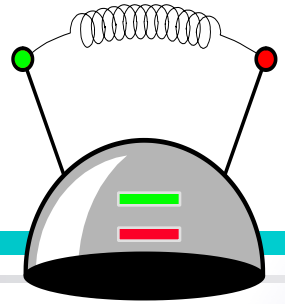
Using the Thinking Cap



- A program that wants to use the thinking cap must **include** the thinker header file (along with its other header inclusions).

```
#include <iostream.h>
#include <stdlib.h>
#include "thinker.h"

...
```

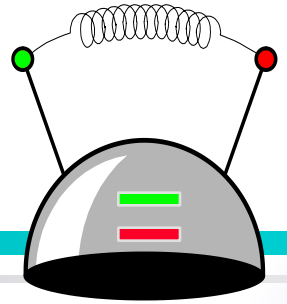
Using the Thinking Cap



- The program starts by calling the slots member function for student.

```
#include <iostream.h>
#include <stdlib.h>
#include "thinker.h"

int main( )
{
    thinking_cap student;
    thinking_cap fan;
    student.slots( "Hello",  "Goodbye");
```

A Quiz

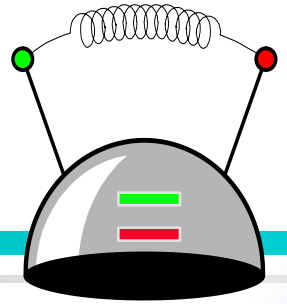


How would you activate student's push_green member function ?

What would be the output of student's push_green member function at this point in the program ?

```
int main( )
{
    thinking_cap student;
    thinking_cap fan;

    student.slots( "Hello",  "Goodbye");

    student.push_green( );
```

A Quiz



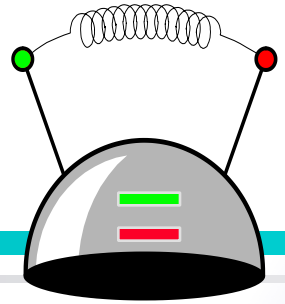
```
int main( )
{
    thinking_cap student;
    thinking_cap fan;
    student.slots( "Hello",  "Goodbye");
    fan.slots( "Go Broncos!", "Boo!");
    student.push_green( );
    fan.push_green( );
    student.push_red( );
    . . .
```

Trace through this program, and tell me the complete output.

What you know about Objects

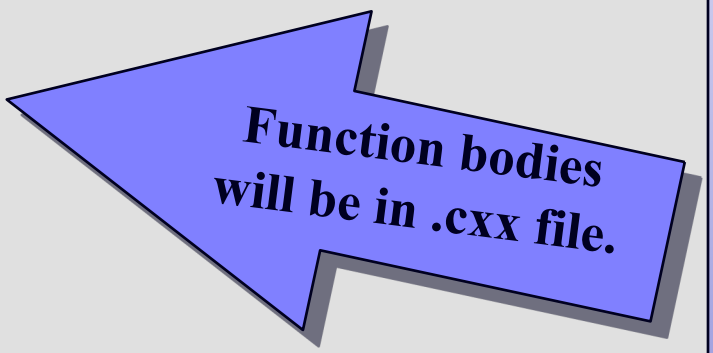
- ✓ Class = Data + Member Functions.
- ✓ You know how to define a new class type, and place the definition in a header file.
- ✓ You know how to use the header file in a program which declares instances of the class type.
- ✓ You know how to activate member functions.
- ✖ But you still need to learn how to write the bodies of a class's member functions.

Thinking Cap Implementation

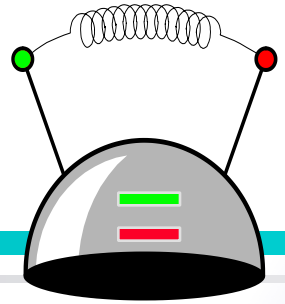


Remember that the member function's bodies generally appear in a separate .cxx file.

```
class thinking_cap
{
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( );
    void push_red( );
private:
    char green_string[50];
    char red_string[50];
};
```



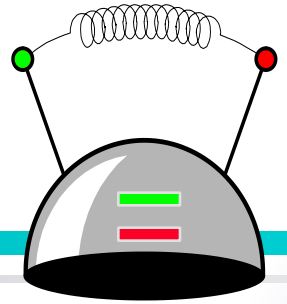
Thinking Cap Implementation



We will look at the body of slots, which must copy its two arguments to the two private member variables.

```
class thinking_cap
{
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( );
    void push_red( );
private:
    char green_string[50];
    char red_string[50];
};
```

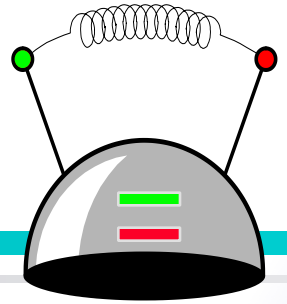
Thinking Cap Implementation



For the most part, the function's body is no different than any other function body.

```
void thinking_cap::slots(char new_green[ ], char new_red[ ])
{
    assert(strlen(new_green) < 50);
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_green);
    strcpy(red_string, new_red);
}
```

Thinking Cap Implementation



- Within the body of the function, the class's member variables and other member functions may all be accessed.

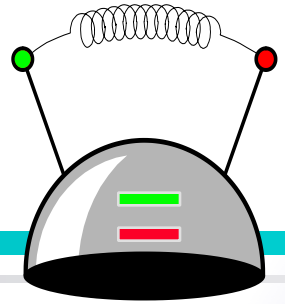
```
void thinking_cap::slots(char new_green[ ], char new_red[ ])
{
    assert(strlen(new_green) < 50);
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_green);
    strcpy(red_string, new_red);
}
```

But, whose member variables are these?

Are they

- ***student.green_string***
- ***student.red_string***
- ***fan.green_string***
- ***fan.red_string***

Thinking Cap Implementation



- Within the body of the function, the class's member variables and other member functions may all be accessed.

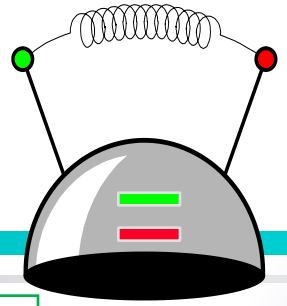
```
void thinking_cap::slots(char new_
{
    assert(strlen(new_green) < 50)
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_gree
    strcpy(red_string, new_red);
}
```

If we activate student.slots:

student.green_string

student.red_string

Thinking Cap Implementation



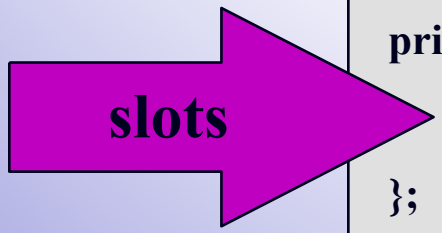
Here is the implementation of the push_green member function, which prints the green message:

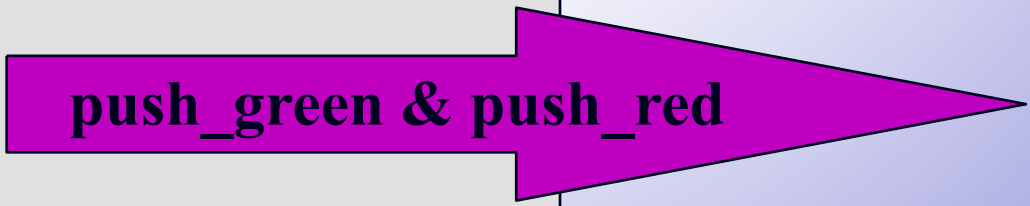
```
void thinking_cap::push_green
{

    cout << green_string << endl;

}
```

A Common Pattern

- Often, one or more member functions will place data in the member variables...

```
class thinking_cap {
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( ) const;
    void push_red( ) const;
private:
    char green_string[50];
    char red_string[50];
};
```





- ...so that other member functions may use that data.

Summary

- Classes have member variables and member functions. An object is a variable where the data type is a class.
- You should know how to declare a new class type, how to implement its member functions, how to use the class type.
- Frequently, the member functions of an class type place information in the member variables, or use information that's already in the member variables.
- In the future we will see more features of OOP.

Class Constructors

- **Constructors** provide an initialization function **that is guaranteed to be called**
- **A constructor is a member function** with **these special properties:**
  - A constructor is **called automatically** whenever a variable of the class is declared
  - The **name of a constructor must be the same as the name of the class**
  - A constructor **does not have any return value**
    - You must not write `void` (or any other return type) at the front of the constructor's head

Question: write the constructor

```
namespace scu_coen79_2A
{
    class point
    {
    public:
        // CONSTRUCTOR
        //????????????????????????????????
        // MODIFICATION MEMBER FUNCTIONS
        void shift(double x_amount, double y_amount);
        void rotate90( );
        // CONSTANT MEMBER FUNCTIONS
        double get_x( ) const { return x; }
        double get_y( ) const { return y; }
    private:
        double x; // x coordinate of this point
        double y; // y coordinate of this point
    };
}
```

Question

- Write a constructor for thinking_cap

```
class thinking_cap {
public:
    void slots(char new_green[ ], char new_red[ ]);
    void push_green( ) const;
    void push_red( ) const;
private:
    char green_string[50];
    char red_string[50];
};
```

Automatic Constructor

If you write a class with no constructor,

- The compiler automatically creates a simple default constructor
- **This automatic default constructor doesn't do much work**

Value Semantic

- The **value semantics** of a class determines how values are copied from one object to another
- In C++, the value semantics consists of two operations:
  - Assignment operator
  - Copy constructor

Assignment Operator

Example:

- DS1 is a data structure that includes the names of employees in a hospital
- DS2 is an object that can include names of employees
- We want to assign the values in DS1 to DS2 This is what we do:

```
employeeName DS2;
DS2 = DS1;
```

- The **automatic assignment operator** simply **copies each member variable** from the object on the right of the assignment to the object on the left of the assignment

Note: Automatic assignment operator **does not always work** (future lectures)

Copy Constructor

Example:

- DS1 is a data structure that includes the names of employees in a hospital
- We want to create an object DS2 while making it an exact copy of DS1

This is what we do:

```
employeeNames DS2(DS1);
```

- **A constructor with exactly one argument**
- The data type of the argument is the same as the constructor's class

Or equivalently:

```
employeeNames DS2 = DS1;
```

C++ Copy Constructor

- C++ provides an **automatic copy constructor**
- The automatic copy constructor initializes a new object by **merely copying all the member variables from the existing object**

- When you implement a class, **the documentation should include a comment indicating that the value semantics is safe to use**

```
// VALUE SEMANTICS    for the XYZ class:
//      Assignments and the copy constructor may be used with XYZ
//      objects.
```

Header files

- What's the problem below?
- What's the solution?

File "a.h"

```
class foo
{
    …
};
```

File "b.h"

```
#include "a.h"
```

File "c.c"

```
#include "a.h"
#include "b.h"
```

Namespace

- Namespaces provide method for preventing name conflicts in large project
- A namespace is a name that a programmer selects to identify a portion of the her work
- A single namespace, such as scu_coen79_2, may have several different namespace groupings
  - e.g, one for class definition, one for implementation

```
namespace scu_coen79_2
{
    //any item that belongs to the namespace is written here.
}
```

Global Namespace & std

The Global Namespace:

- Any items that are not explicitly placed in a namespace become part of the global namespace
  - These items can be used at any point without any need for a `using` statement or a scope resolution operator

C++ Standard Library (`std` namespace):

- All of the items in the C++ Standard Library are automatically part of the `std` namespace
  - When using C++ header file names

    i.e., `<iostream>` or `<cstdlib>`
  - The simplest way is to add "`using namespace std;`"

Using Namespaces

How to use the items that are defined in the namespace:

1. Make all of the namespace available

   Syntax: **`using namespace`** `ns_name;`

   - Example: `using namespace scu_coen79_2A;`
     - Question: Why is this a bad idea to put a `using` statement in a header file?

2. If we need to **use only a specific item from the namespace:**

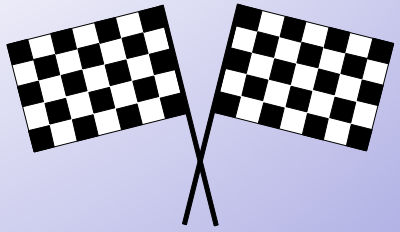
   Syntax: **`using ns_name::name;`**

   - Example: `using scu_coen79_2A::throttle;`

3. Use any item by **prefixing the item name with the namespace and "::"** at the point where the item is used

   Syntax: **`ns_name::name`**

   - Example: `scu_coen79_2A::throttle apollo;`



THE END