C `struct` — Enables us to...

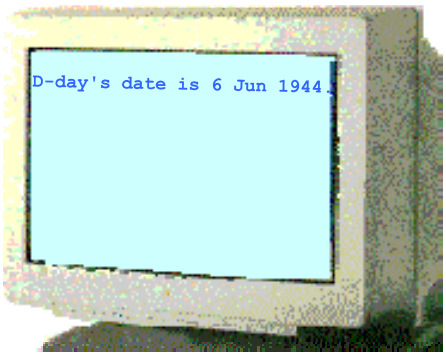
- Group related data components of *different types* under one name (*heterogeneous*)
  - ***array*** ➔ group related data components of the *same type* under one name (*homogeneous*)
  - ***array*** and **`struct`** can be used together ➔ such as an array of **`struct`**'s and having an array as a member of a **`struct`**
- Define custom *composite data types* from existing types
- For example, we can define a custom composite data type called **`StudentInfo`** that is comprised of
  - **`studentId`** – perhaps an **`int`**
  - **`studentName`** – perhaps a C-string
  - **`studentGpa`** – perhaps a **`double`**
  - `...`

C `struct` — Example – Defining Date

```
struct Date
{
   int day;       // valid range: 1 – 31
   char month[4]; // valid range: "Jan" through "Dec"
   int year;      // e.g. 1999
};
```

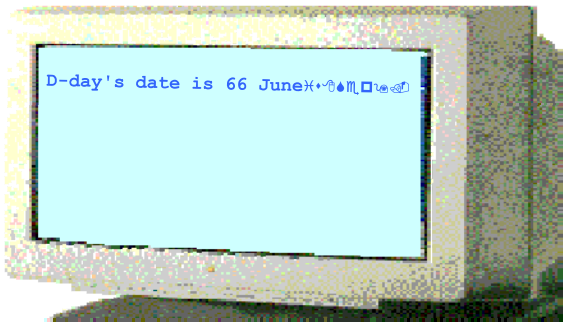
C **struct** Example – Using Date

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

int main()
{
   Date dDay;

   dDay.year = 1944;
   strcpy(dDay.month, "Jun");
   dDay.day = 6;
   cout << "D-day's date is " << dDay.day << ' '
        << dDay.month << ' ' << dDay.year << '.' << endl;

   return EXIT_SUCCESS;
}
```



C **struct** Not too excited with this blind Date?

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

int main()
{
   Date dDay;

   dDay.year = 1944;
   strcpy(dDay.month, "June"); // array bound exceeded
   dDay.day = 66;              // invalid day
   cout << "D-day's date is " << dDay.day << " "
        << dDay.month << " " << dDay.year << "." << endl;

   return EXIT_SUCCESS;
}
```



C struct

Would these chaperons help?

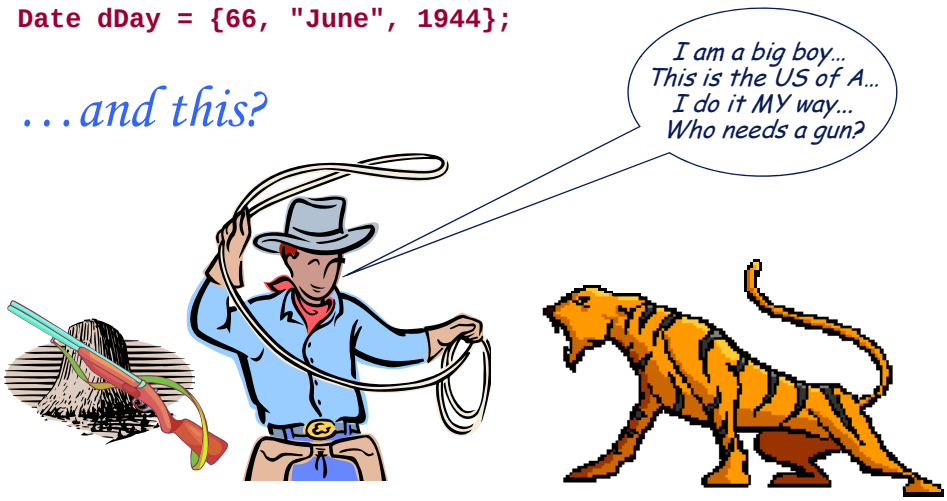
```
void SetYear(Date& d)
{
   ...
}

void SetMonth(Date& d)
{
   ...
}

void SetDay(Date& d)
{  // NOTE: much more elaborate checking can be effected
   //       (e.g., take year & month into consideration)
   int dayInput;
   cout << "Enter day (1 – 31): ";
   cin >> dayInput;
   if (dayInput > 0 && dayInput < 32)
      d.day = dayInput;
   else
      cerr << "Invalid day." << endl;
}
```


C struct

Help? Yes. Guarantee? No.

```
// What about, say, this?
Date dDay = {66, "June", 1944};
```

. . . and this?



C `struct` Problem

- Data *not protected* against misuse or abuse
  - C ➔ ***procedural paradigm*** – lack of data protection is not unexpected: actions take center stage, data plays supporting role
  - lack of data protection is actually only a small part of a bigger problem – user is exposed to how data is implemented and allowed/required to manipulate data directly ➔ in general, there's a lack of emphasis on how data should be treated and presented (to the user)
  - *lack of emphasis on data* – main cause of problems associated with debugging and maintaining *large* programs
- Realization of the preceding problem coupled with the anticipation of an ever-increasing trend in program size (➔ the *software crisis*) has led to a shift in paradigm
  - first from procedural to ***object-based***
  - then to ***object-oriented***

Object–Based Paradigm Key Feature

Overcomes the procedural paradigm's lack of emphasis on data by providing a mechanism that enables programmers to *selectively restrict access to data*

- through ***data encapsulation*** – by packaging data and associated operations into unified entities (objects)
- promotes ***data/information hiding*** – user is shielded from data implementation and allowed to manipulate data only through some well-defined and well-behaved interfaces
- a benefit of data/information hiding is that, by restricting the user from directly accessing data, data is protected from getting accidentally or maliciously corrupted
- but data/information hiding, when properly applied, is key to high quality code and software (better maintainability, better updatability/upgradeability, *etc.*)

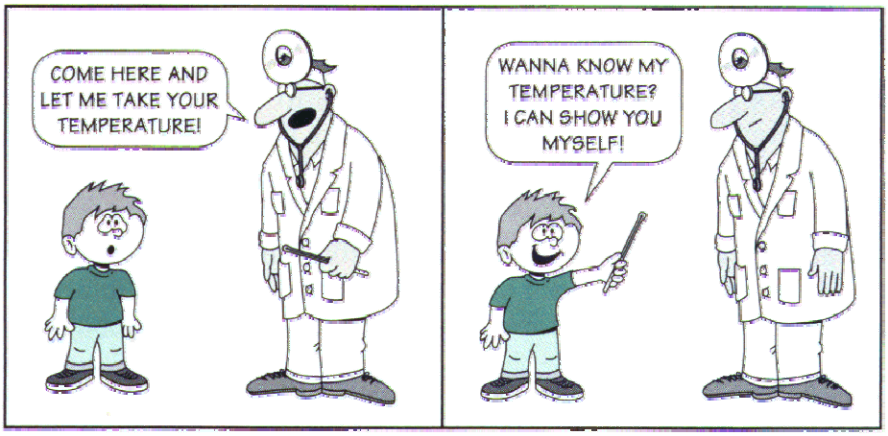
In the context of... our **`Date`**

We would like to have a mechanism whereby the users (clients) of **`Date`**...

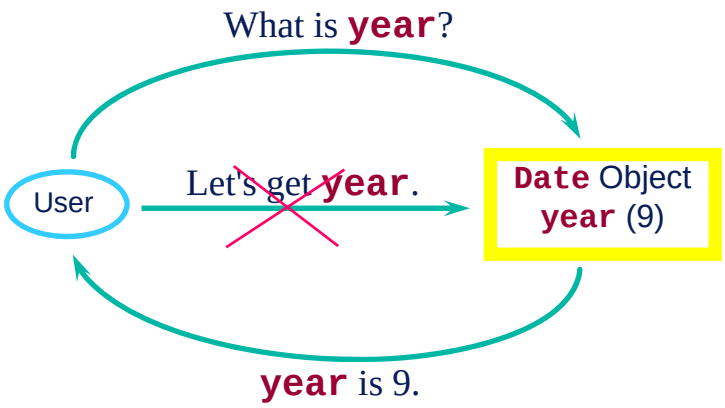
- *cannot directly access* the data members **`day`**, **`month`** and **`year`**, but instead ...
- are allowed to *access them only indirectly* through the *functions that we provide*, such as **`SetDay()`**, **`SetMonth()`**, **`SetYear()`**, **`GetDay()`**, **`GetMonth()`**, **`GetYear()`**, **`ShowDay()`**, **`ShowMonth()`**, **`ShowYear()`**, and so on
- (the above functions actually provides a rather ***low*** level of data/information hiding – what functions would you suggest, in place of those above, that would provide a ***higher*** level of data/information hiding?)

Procedural vs Object-Based Environment

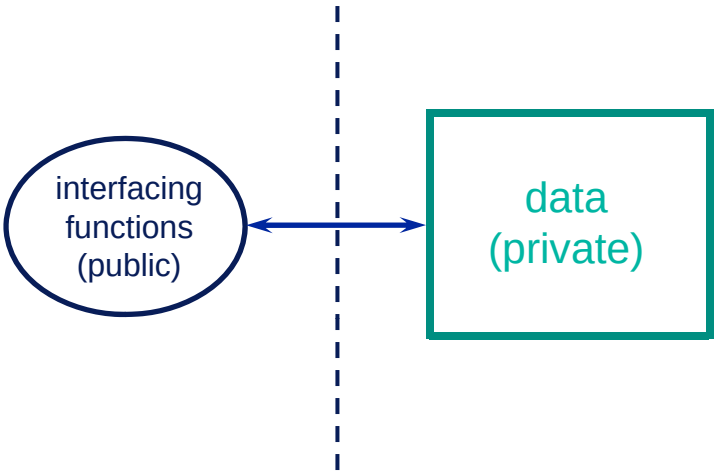
("procedural" attempt) ("object–based" response)



Object-Based Environment



Data/Information Hiding



In other words... we'd like our **`Date`** to be like

```
struct Date
{
   private: // data are not publicly accessible

      int day;       // valid range: 1 – 31
      char month[4]; // valid range: "Jan" through "Dec"
      int year;      // e.g. 1999

   public:  // interfacing functions are publicly accessible

      void SetDay(...)
      {
         ...
      }

      void SetMonth(...)
      {
         ...
      }

      ...  // more public functions here
};
```


Perhaps... it also deserves a more **`class`**y name

```
class Date
{
   private: // data are not publicly accessible

      int day;       // valid range: 1 – 31
      char month[4]; // valid range: "Jan" through "Dec"
      int year;      // e.g. 1999

   public:  // interfacing functions are publicly accessible

      void SetDay(...)
      {
         ...
      }

      void SetMonth(...)
      {
         ...
      }

      ...  // more public functions here
};
```

`private` and **`public`** are called ***member access specifiers*** ➔ there is a third one called **`protected`** that comes into prominence (with ***inheritance***) under *object-oriented* (not *object-based*) paradigm

And behold... C++ `class`

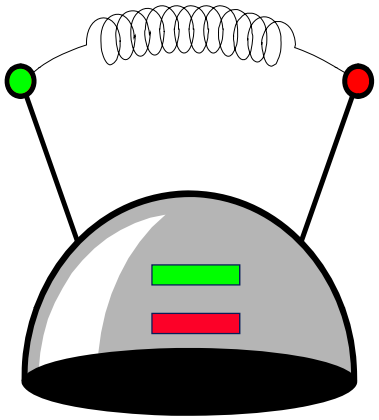
- Call that the birth story of C++ **`class`** if you will
- Of course, it has since matured into something much more than its humble beginning
- In C++, **`class`** and **`struct`** differ only in that
  - the default access specifier for **`struct`** is ***public***, and
  - the default access specifier for **`class`** is ***private***
- But not C **`struct`**

And lest you should think otherwise...

- The C++ **`class`** mechanism offers a lot more than just ***data protection*** that we have discussed
- Some of these (definitely not all) are the subjects of our study to come
- But let us first look at a simple example of **`class`** in action and...
  - ...preview some associated issues/ideas/"how-to"s

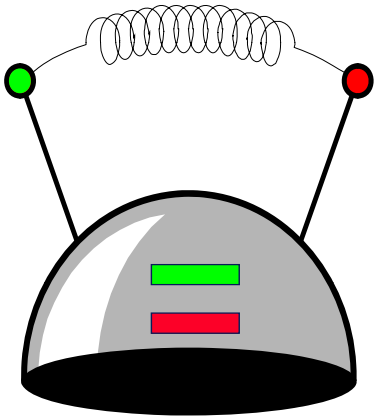
What is this Object ?

- 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

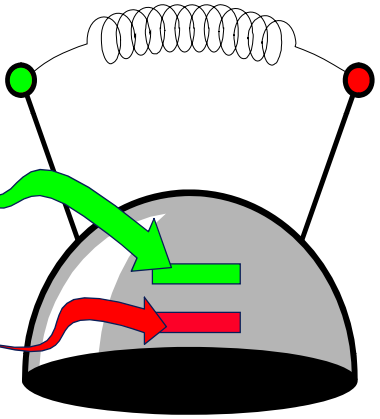
- You may put a piece of paper in each of the two slots (green and red), with a sentence written on each.
- You may push the green button and the thinking cap will speak the sentence from the green slot's paper.
- And same for the red button.



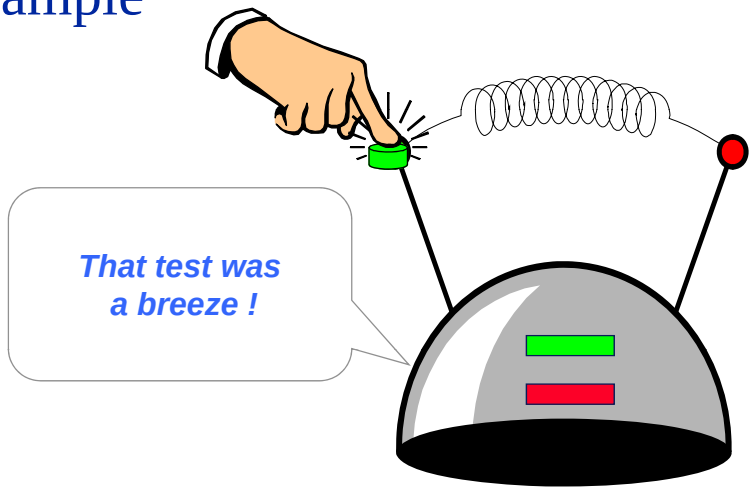
Example

That test was a breeze !

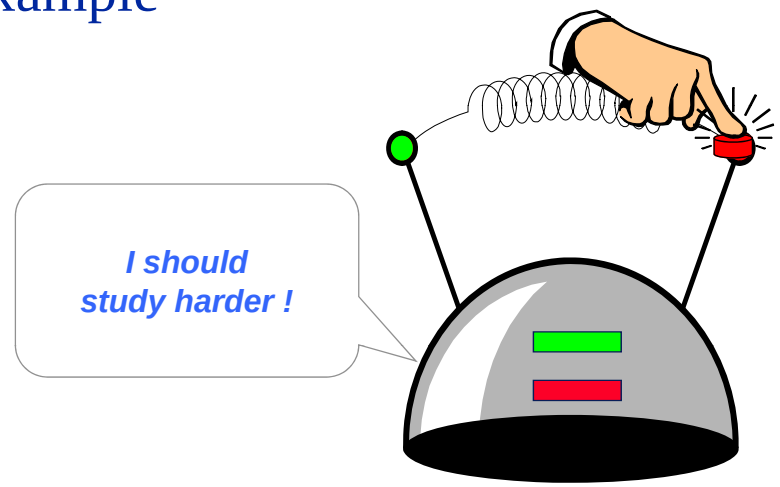
I should study harder !



Example



Example

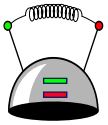


Thinking Cap Implementation

- We can use **`class`** to implement the thinking cap

```
class ThinkingCap
{

   . . .

};
```

Thinking Cap Implementation



- The class will have two components called **`green_string`** and **`red_string`**. These components are strings (C-style) which hold the information that is placed in the two slots.
- Using a class permits *two new features* . . .

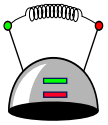
```
class ThinkingCap
{
   . . .
   char green_string[51];
   char red_string[51];
   . . .
};
```


Thinking Cap Implementation

- The two components will be ***private member variables***. This ensures that nobody can directly access this information. The only access is through functions that we provide for the class.

```
class ThinkingCap
{
   . . .
private:
   char green_string[51];
   char red_string[51];
};
```

Thinking Cap Implementation



- In a class, the functions which manipulate the class are also listed.

Prototypes for the thinking cap functions go here, after the label **`public:`**

```
class ThinkingCap
{
public:
   . . .
private:
   char green_string[51];
   char red_string[51];
};
```

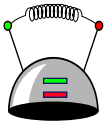

Thinking Cap Implementation

- In a class, the functions which manipulate the class are also listed.

Prototypes for the thinking cap ***member functions*** go here

```
class ThinkingCap
{
public:
   . . .
private:
   char green_string[51];
   char red_string[51];
};
```

Thinking Cap Implementation



Our thinking cap has at least 3 member functions:

```
class ThinkingCap
{
public:
   void slots(char new_green[], char new_red[]);
   void push_green() const;
   void push_red() const;
private:
   char green_string[51];
   char red_string[51];
};
```

Function bodies will be elsewhere.

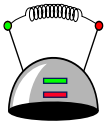
Thinking Cap Implementation

The keyword **`const`** appears after two prototypes:

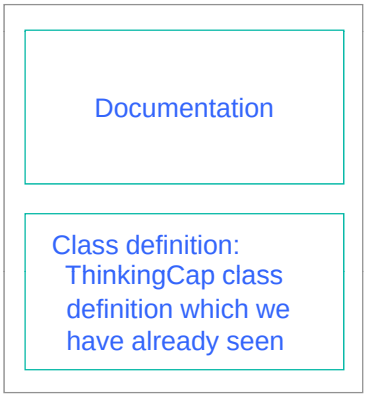
```
class ThinkingCap
{
public:
   void slots(char new_green[], char new_red[]);
   void push_green() const;
   void push_red() const;
private:
   char green_string[51];
   char red_string[51];
};
```

`const` specifies that the function shall not change the data of the **activating ThinkingCap object**.

Files for the Thinking Cap



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

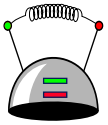


Using the Thinking Cap

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

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

Using the Thinking Cap



- Just for fun, the example program will declare two **`ThinkingCap`** variables named **`student`** and **`fan`**.

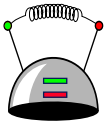
```
#include <iostream>
#include <cstdlib>
#include "thinker.h"
using namespace std;

int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   ...
```


Using the Thinking Cap

- Just for fun, the example program will declare two **`ThinkingCap`** ***objects*** named **`student`** and **`fan`**.

```
#include <iostream>
#include <cstdlib>
#include "thinker.h"
using namespace std;

int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   ...
```

Using the Thinking Cap



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

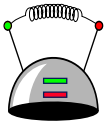
```
#include <iostream>
#include <cstdlib>
#include "thinker.h"
using namespace std;

int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   student.slots("Hello", "Bye");
   ...
```


Using the Thinking Cap

- The program starts by *activating* the slots *member function* for **`student`**.

```
#include <iostream>
#include <cstdlib>
#include "thinker.h"
using namespace std;

int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   student.slots("Hello", "Bye");
   ...
```

Using the Thinking Cap



- The member function activation consists of four parts, starting with the object name.

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

   student.slots("Hello", "Bye");
   ...
```

Name of the object

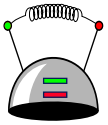
Using the Thinking Cap

- The instance name is followed by a period, which is the ***dot operator***.

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

   student.slots("Hello", "Bye");
   ...
```

A period

Using the Thinking Cap



- After the period is the name of the member function that you are activating.

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

   student.slots("Hello", "Bye");
   ...
```

Name of the function

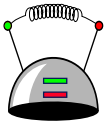
Using the Thinking Cap

- Finally, the arguments for the member function. In this example the first argument (**new_green**) is **"Hello"** and the second argument (**new_red**) is **"Bye"**.

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

   student.slots("Hello", "Bye");
```

Arguments

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()
{
   ThinkingCap student;
   ThinkingCap fan;

   student.slots("Hello", "Bye");
   ...
```

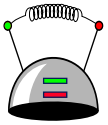

A Quiz

Notice that the **push_green** member function has no arguments.

At this point, activating **student.push_green** will print the string **Hello**.

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

   student.slots("Hello", "Bye");
   student.push_green();
   …
```

A Quiz



```
int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   student.slots("Hello", "Bye");
   fan.slots("Go Bobcats!", "Boo!");
   student.push_green();
   fan.push_green();
   student.push_red();
   . . .
```

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

A Quiz

```
int main()
{
   ThinkingCap student;
   ThinkingCap fan;
   student.slots("Hello", "Bye");
   fan.slots("Go Bobcats!", "Boo!");
   student.push_green();
   fan.push_green();
   student.push_red();
   . . .
```

Hello
Go Bobcats!
Bye

What you know by now ...

- 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 **.cpp** file.

```
class ThinkingCap
{
public:
   void slots(char new_green[], char new_red[]);
   void push_green() const;
   void push_red() const;
private:
   char green_string[51];
   char red_string[51];
};
```

Function bodies will be in .cpp file.

Thinking Cap Implementation

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

```
class ThinkingCap
{
public:
   void slots(char new_green[], char new_red[]);
   void push_green() const;
   void push_red() const;
private:
   char green_string[51];
   char red_string[51];
};
```

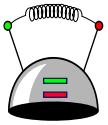

Thinking Cap Implementation

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

```
void ThinkingCap::slots(char new_green[], char new_red[])
{
     assert(strlen(new_green) < 51);
     assert(strlen(new_red) < 51);
     strcpy(green_string,  new_green);
     strcpy(red_string, new_red);
}
```

But there are two special features about a member function's body . . .

Thinking Cap Implementation



- In the heading, the function's name is preceded by the **class name** and **::** – otherwise C++ won't realize this is a class's member function.

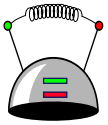
```
void ThinkingCap::slots(char new_green[], char new_red[])
{
     assert(strlen(new_green) < 51);
     assert(strlen(new_red) < 51);
     strcpy(green_string, new_green);
     strcpy(red_string, new_red);
}
```


Thinking Cap Implementation

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

```
void ThinkingCap::slots(char new_green[], char new_red[])
{
     assert(strlen(new green) < 51);
     assert(strlen(new_red) < 51);
     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 ThinkingCap::slots(char
{
     assert(strlen(new green
     assert(strlen(new_red) <
     strcpy(green_string, new
     strcpy(red_string, new_r
}
```

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.

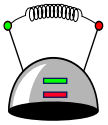
```
void ThinkingCap::slots(char
{
     assert(strlen(new green
     assert(strlen(new_red) <
     strcpy(green_string, new
     strcpy(red_string, new_r
}
```

If we activate `student.slots`***:***

`student.green_string`

`student.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 ThinkingCap::slots(char
{
     assert(strlen(new green
     assert(strlen(new_red) <
     strcpy(green_string, new
     strcpy(red_string, new_r
}
```
)

If we activate **`fan.slots`***:*

```
fan.green_string
fan.red_string
```

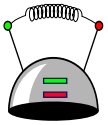
- How do they do it? → the **`this`** pointer

Thinking Cap Implementation

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

```
void ThinkingCap::push_green() const
{
   cout << green_string << endl;
}
```

Thinking Cap Implementation



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

```
void ThinkingCap::push_green() const
{
   cout << green_string << endl;
}
```

Notice how this member function implementation uses the **green_string** member variable of the object.

A Common Pattern

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

```
class ThinkingCap
{
public:
   void slots(char new_green[], char new_red[]);
   void push_green() const;
   void push_red() const;
private:
   char green_string[51];
   char red_string[51];
};
```

slots

push_green and **push_red**

- … 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 ➔ an instance of 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 a class type place information in the member variables, or use information that's already in the member variables.

Textbook Readings

- Chapter 2
  - Section 2.1
  - Section 2.3 (including the discussions on ***namespace***)
  - Section 2.4 (if you are not already familiar with ***passing by value***, ***passing by reference***, and ***passing by `const` reference***)