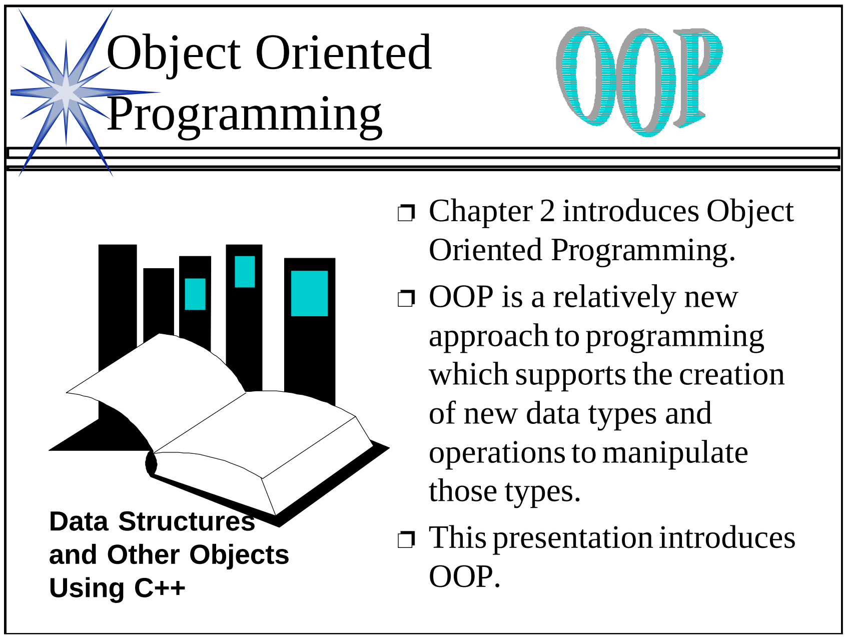


Object Oriented Programming

OOP

- Chapter 2 introduces Object Oriented Programming.
- OOP is a relatively new approach to programming which supports the creation of new data types and operations to manipulate those types.
- This presentation introduces OOP.

Data Structures and Other Objects Using C++

This lecture is an introduction to classes, telling what classes are and how they are implemented in C++. The introduction is basic, not covering constructors or operators that are covered in the text. The best time for this lecture is just before students read Chapter 2--perhaps as early as the second day of class.

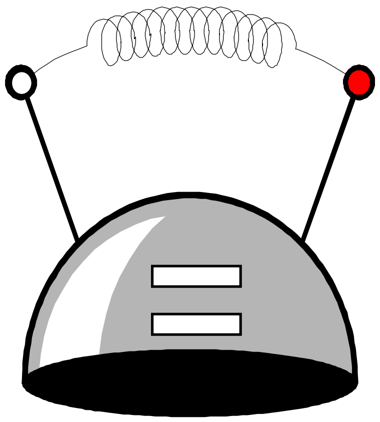
Before this lecture, students should have a some understanding of

1. How an array of characters can be used as a string in C++ programming, and

2. The meaning of the strlen and strcpy functions from string.h.

What is this Object ?

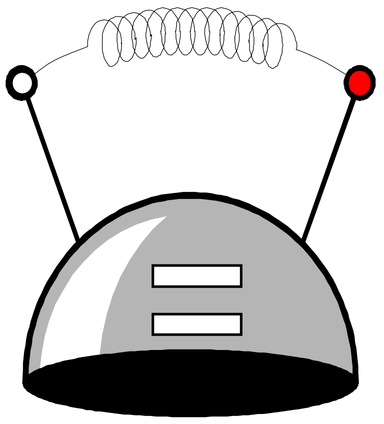
- There is no real answer to the question, but we'll call it a "thinking cap".
- The plan is to describe a thinking cap by telling you what actions can be done to it.



This lecture will introduce you to object-oriented programming by using one example, which we'll call a "thinking cap".

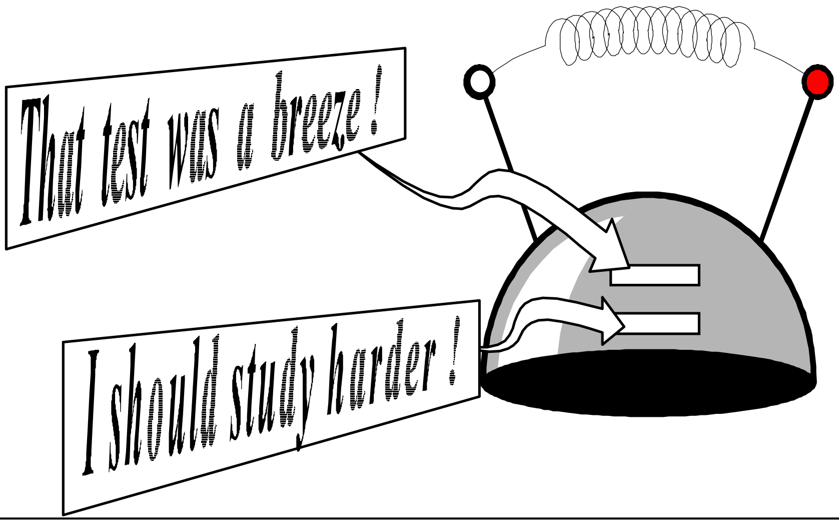
Using the Object’s Slots

- ❒ 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.

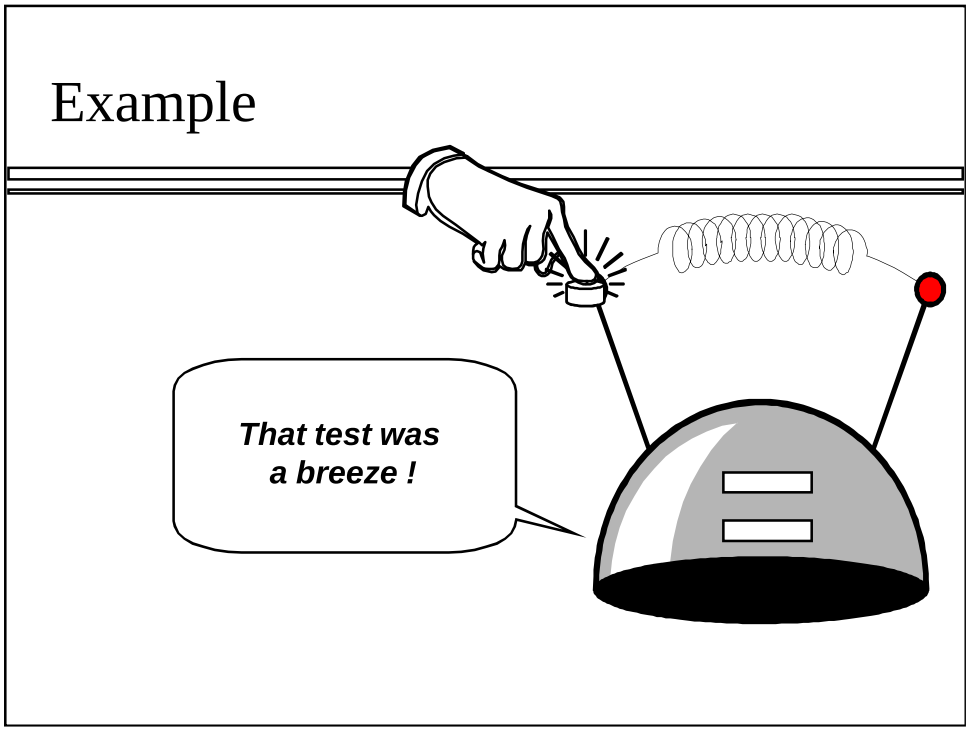


The important thing about this thinking cap is that there are three actions which may happen to it. The three actions are described here.

Example



Here's an example of how the first action works. Messages are written on two slips of paper, and the messages are inserted in the two slots.

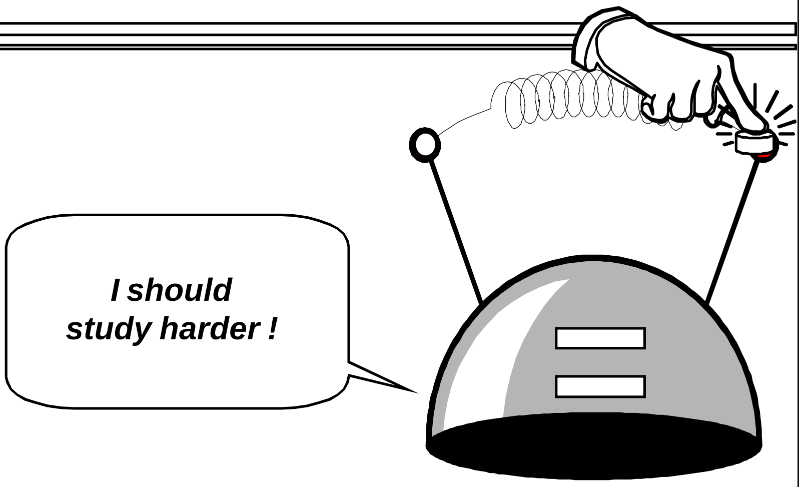


Example

That test was a breeze !

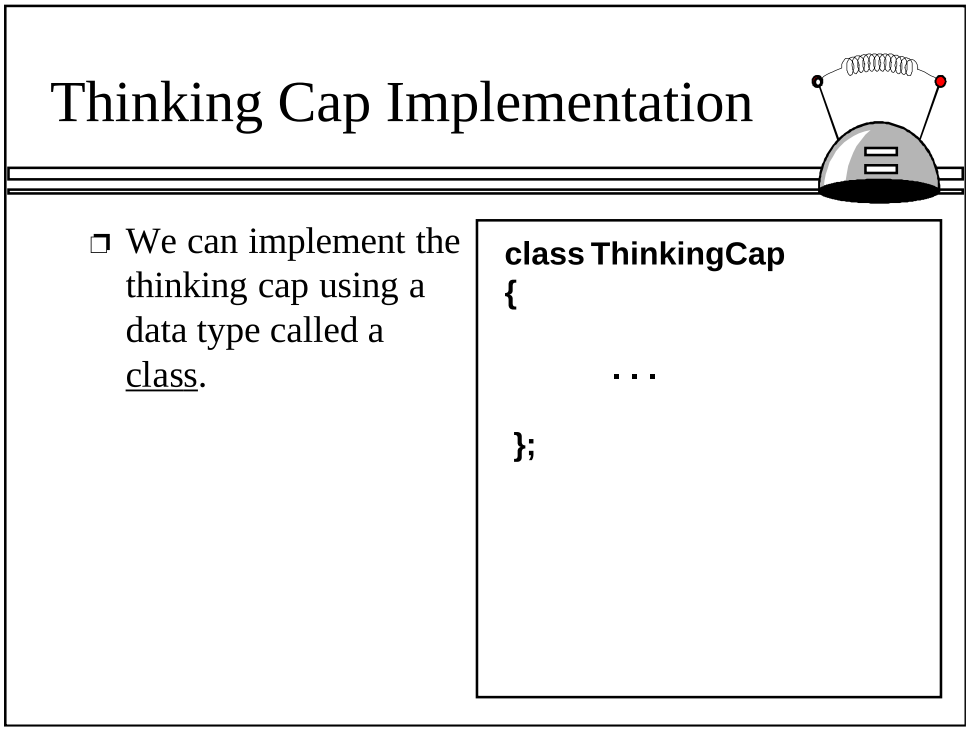
Once the messages have been inserted, either of the buttons may be pressed. When the green button is pressed, the message from the green slip of paper is spoken.

Example



When the red button is pressed, the message from the red slip of paper is spoken.

By the way, what would be an appropriate precondition for pressing the red button? Answer: Before the button is pressed, the slips of paper should be inserted in the slots.

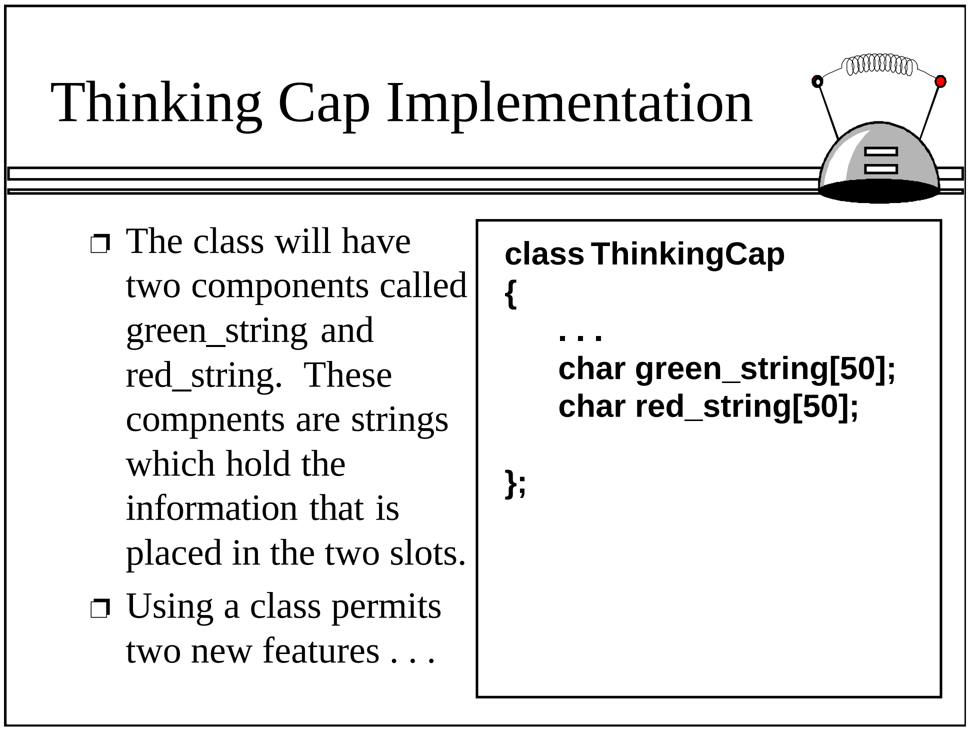


Thinking Cap Implementation

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

```
class ThinkingCap
{

        . . .

 };
```

We will implement the thinking cap in C++ using a feature called a class.



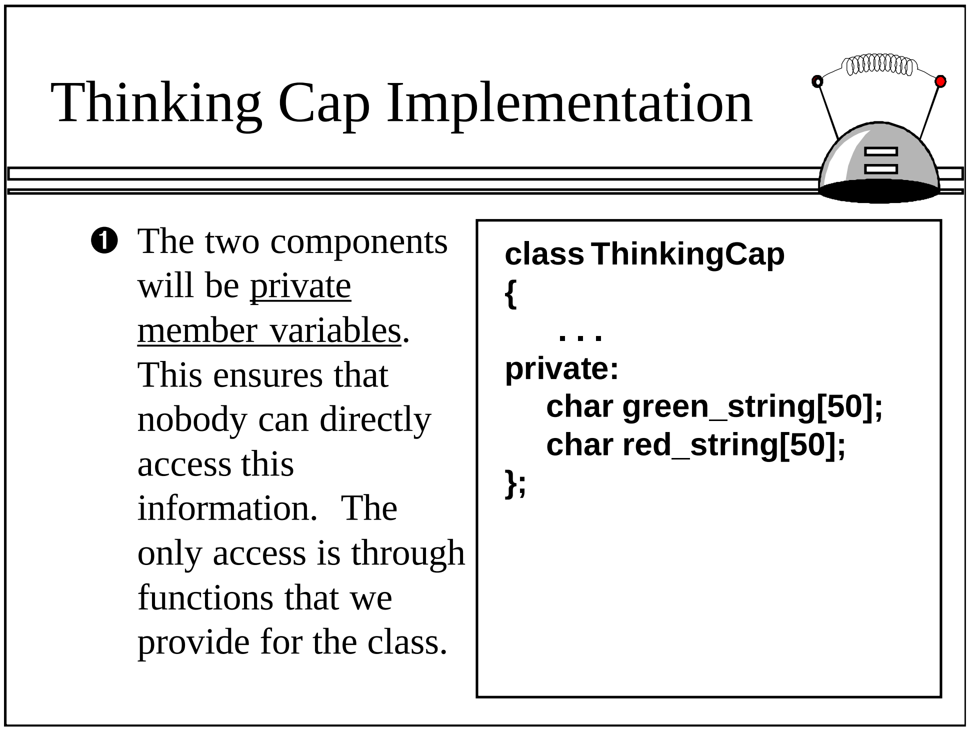
Thinking Cap Implementation

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

```
class ThinkingCap
{
    . . .
    char green_string[50];
    char red_string[50];

};
```

The particular class we have in mind has two components that store the two sentences that are inserted in those slots. These components can be declared arrays of characters in C++. As you may know, an array of character can be used in C++ to store a string. In this case, the string may be up to 49 characters (because we must save at least one spot for the “end of string” marker).

Some of you may have used classes before in your programming. Others might have used “structs”, which are similar to classes. But a C++ class has two new features that are not available in ordinary struct types...

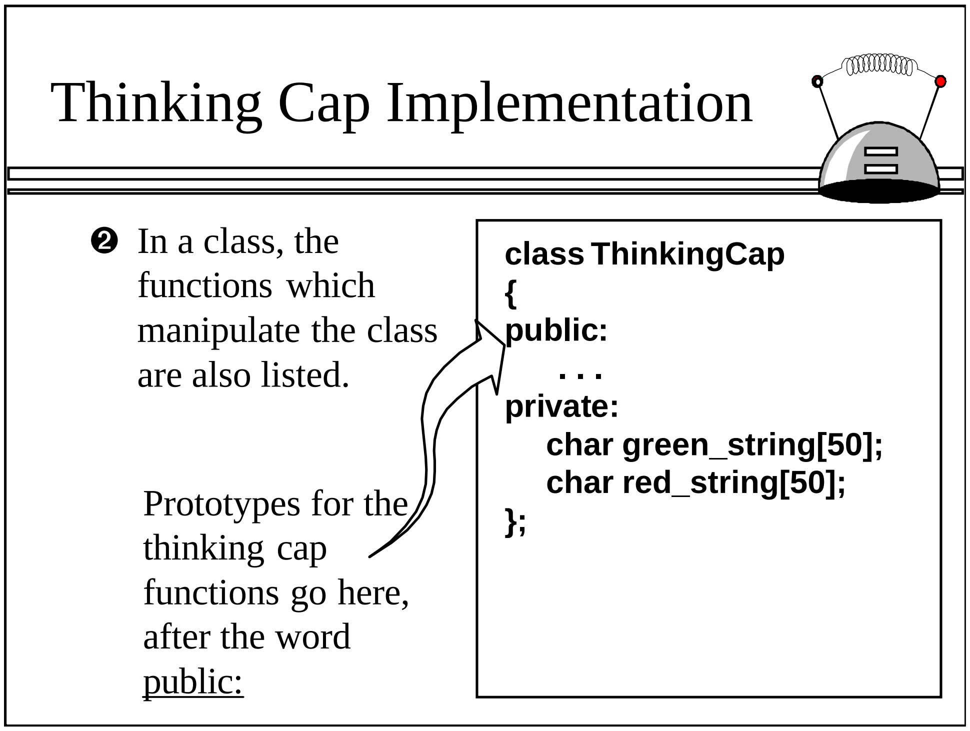


Thinking Cap Implementation

❶ The two components will be <u>private member variables</u>. 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[50];
    char red_string[50];
};
```

The first class feature is that class components are allowed to be <u>private</u> components. The advantage of private components is that they prevent certain programmers from accessing the components directly. Instead, programmers are forced to use only through the operations that we provide.



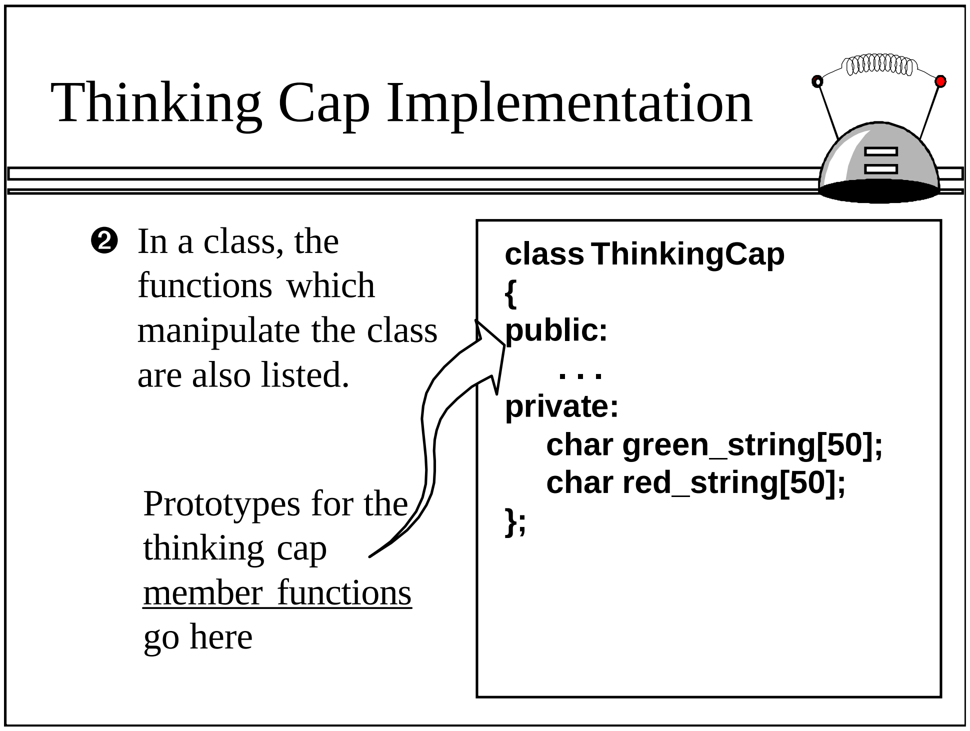
Thinking Cap Implementation

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

```
class ThinkingCap
{
public:
     . . .
private:
    char green_string[50];
    char red_string[50];
};
```

Prototypes for the thinking cap functions go here, after the word public:

In a class, the operations to manipulate the data are actually part of the class itself. A prototype for each function is placed as part of the class definition.



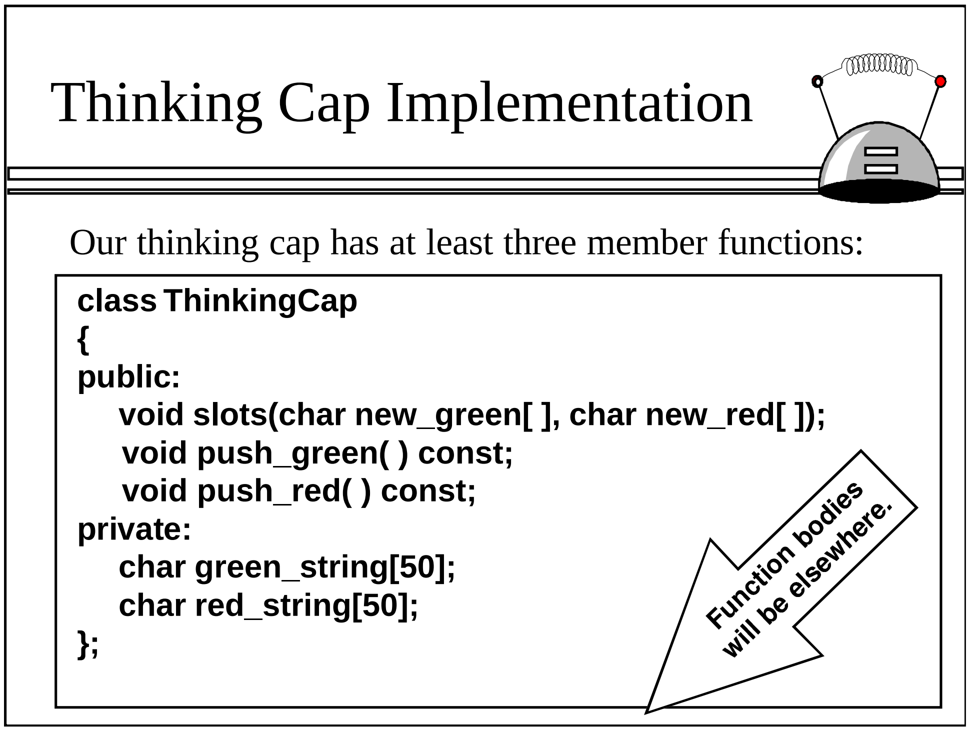
Thinking Cap Implementation

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

```
class ThinkingCap
{
public:
      . . .
private:
    char green_string[50];
    char red_string[50];
};
```

Prototypes for the thinking cap <u>member functions</u> go here

In the jargon of OOP programmers, the class’s functions are called it’s <u>member functions</u>, to distinguish them from ordinary functions that are not part of a class.



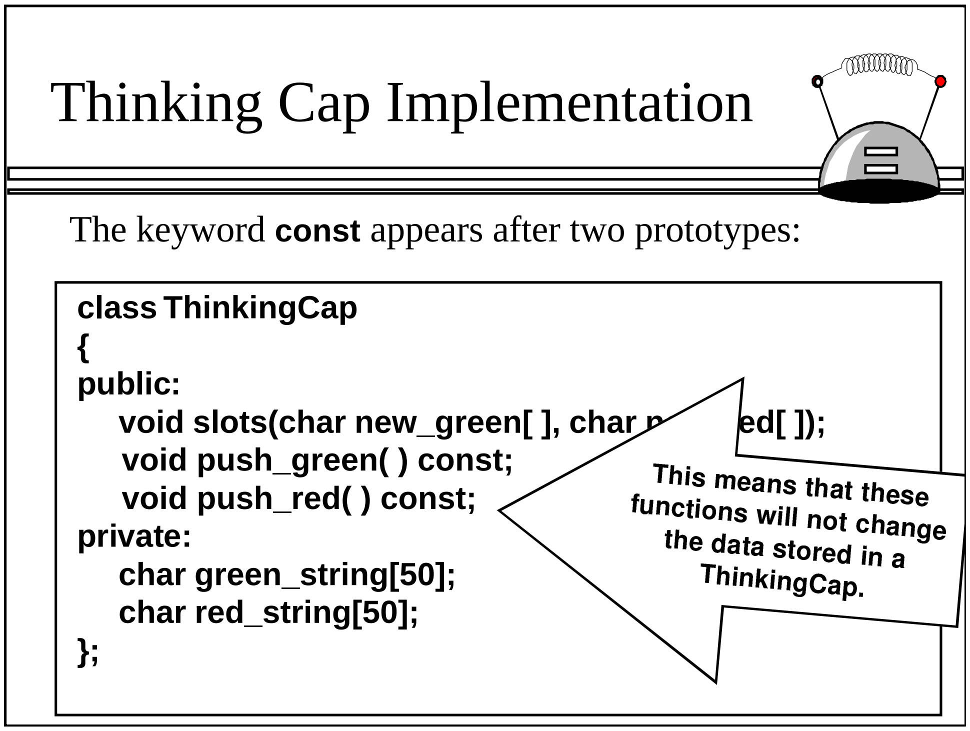
Thinking Cap Implementation

Our thinking cap has at least three 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[50];
    char red_string[50];
};
```

Function bodies will be elsewhere.

The implementations of member functions do not normally appear withing the class definition. We’ll see where they do appear later, but for now, let's just concentrate on this part of the class, which is called the <u>class definition</u>.



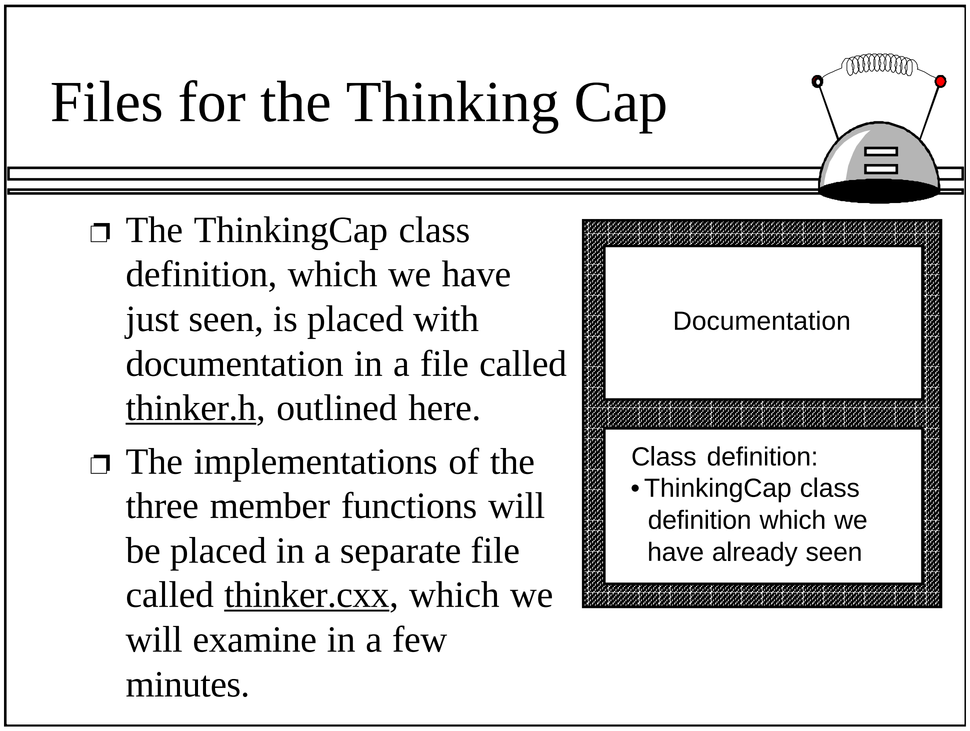
Thinking Cap Implementation

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

```
class ThinkingCap
{
public:
    void slots(char new_green[ ], char n[illegible]ed[ ]);
     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 ThinkingCap.

One thing that you might have noticed in the definition is a keyword, const, which appears after two of my prototypes. This keyword means that these two functions will not change the data stored in a ThinkingCap. In other words, when you do use these two functions, a ThinkingCap remains “constant”.



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.cxx, which we will examine in a few minutes.

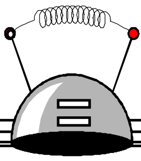
Documentation

Class definition:
- ThinkingCap class definition which we have already seen

Typically, a class definition is placed in a separate header file along with documentation that tells how to use the new class. The implementations of the member functions are placed in a separate file called the implementation file.

At this point, I still haven't shown you exactly what those three implementations of member functions look like -- and I want to continue to postpone that. Instead, I will next show you an example program which uses this ThinkerCap class.

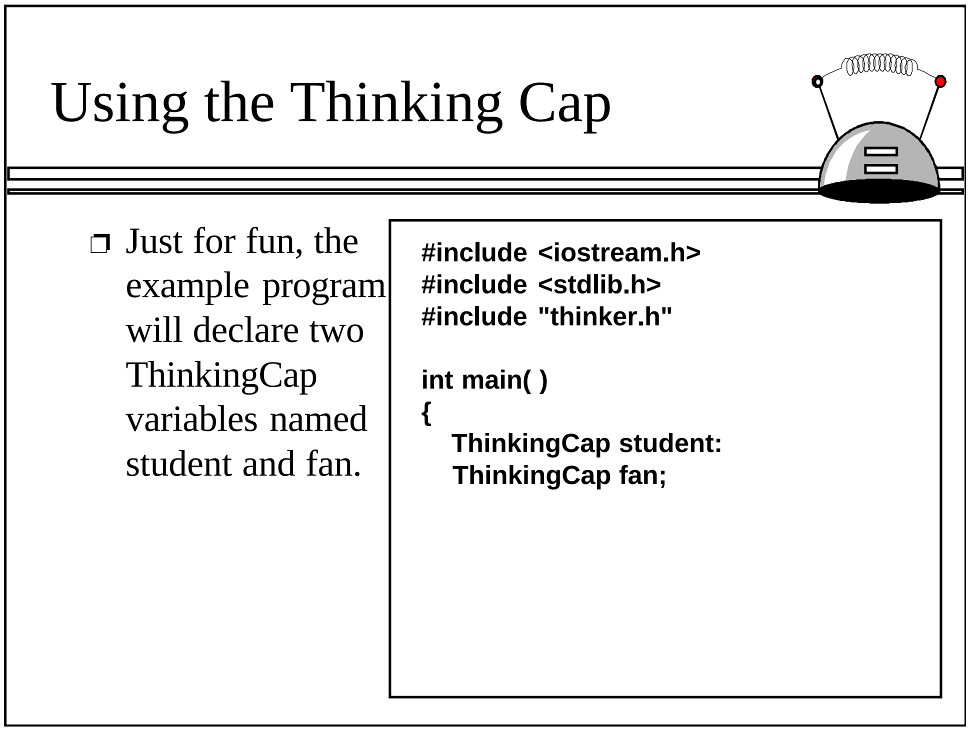
Using the Thinking Cap



- 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"

...
```

Any program that uses a class requires an include statement indicating the name of the header file that has the class definition. Note that we include only thinker.h, which is the header file, and do not include the implmentation file.



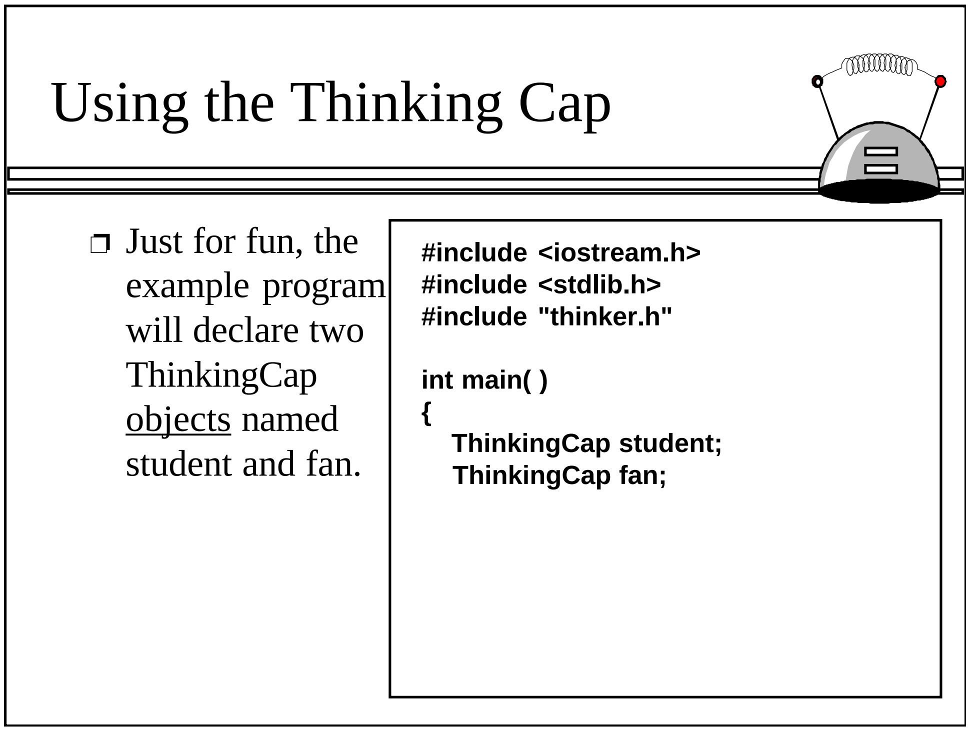
Using the Thinking Cap

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

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

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

After the include statement, we may declare and use variables of the ThinkingCap data type.

This example actually has two ThinkingCap variables, named student and fan.

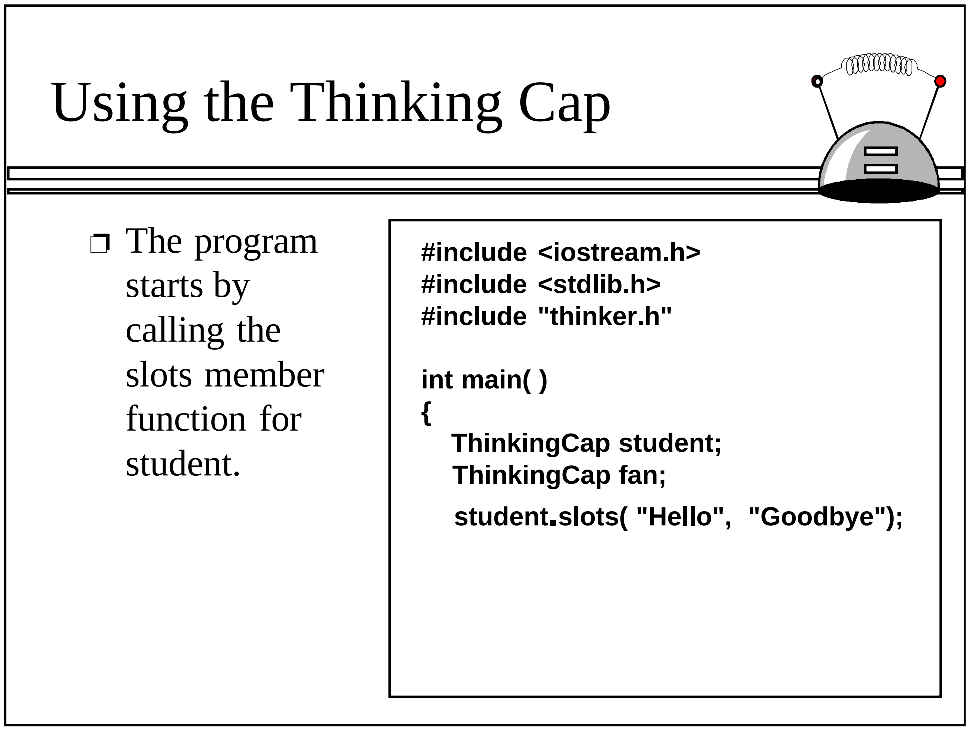


Using the Thinking Cap

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

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

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

In object-oriented terminology, we would call these two variables objects of the ThinkingCap class.

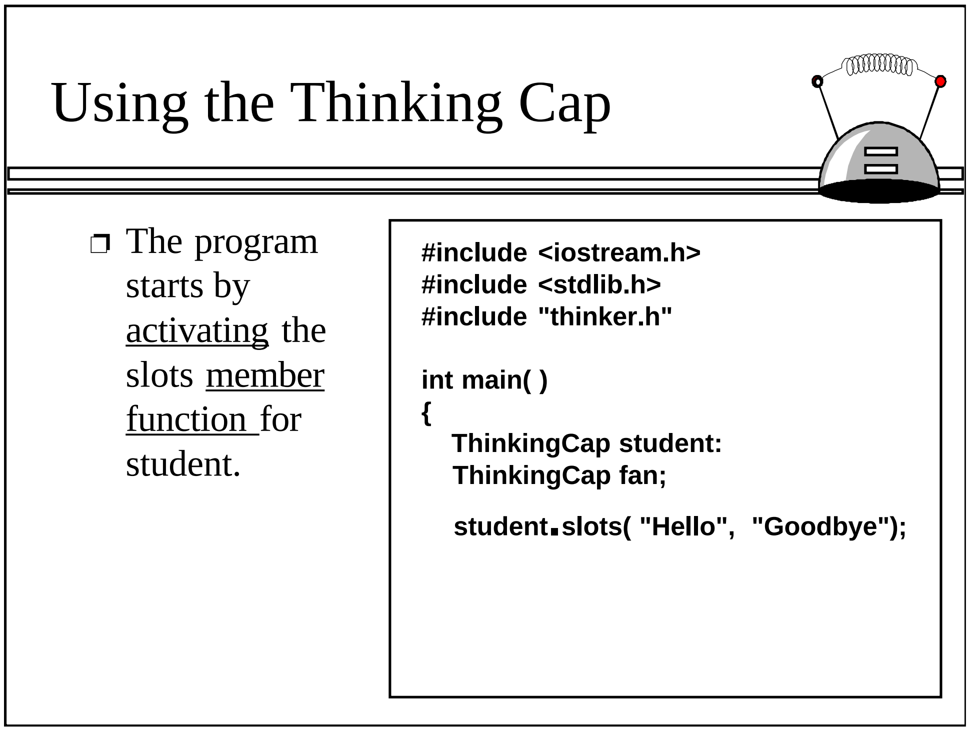


Using the Thinking Cap

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

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

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

This illustrates how we call one of the ThinkingCap functions for the student object.

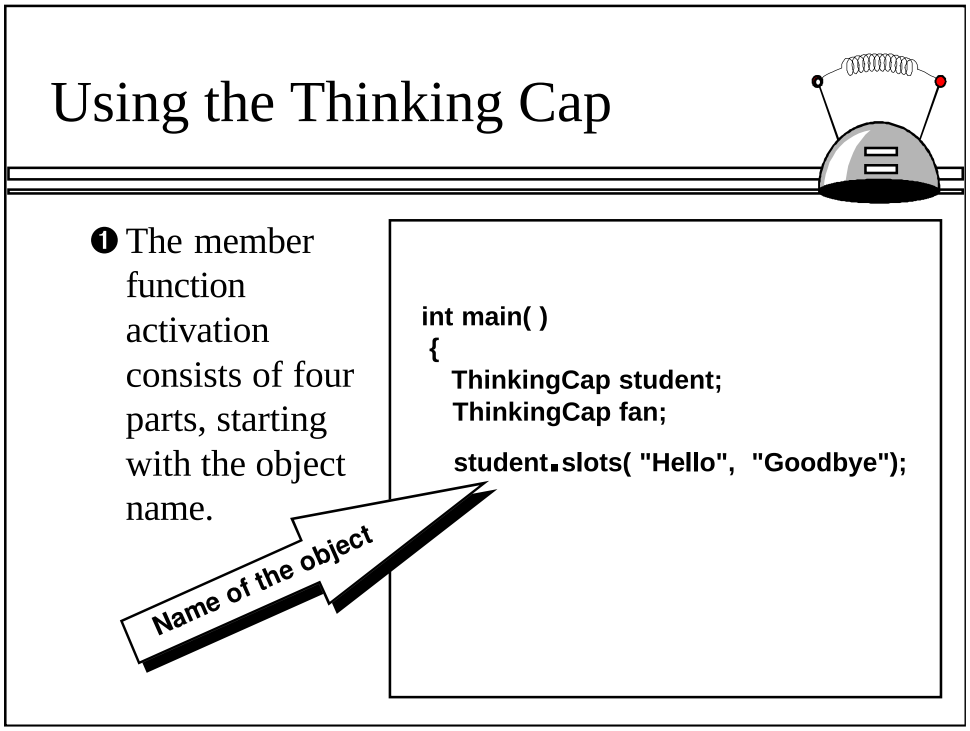


Using the Thinking Cap

- The program starts by <u>activating</u> the slots <u>member function</u> for student.

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

int main( )
{
   ThinkingCap student:
   ThinkingCap fan;

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

But, again, let's use the usual OOP terminology, so that instead of saying that we are calling a function we say that we are <u>activating a member function</u>. In particular, we are activating the slots member function of the student object. (If you go to a cocktail party and tell your OOP friends that today you called a function for an object, they will laugh behind your back. It is better to impress them by saying that you activated a member function, even though it’s just jargon.)



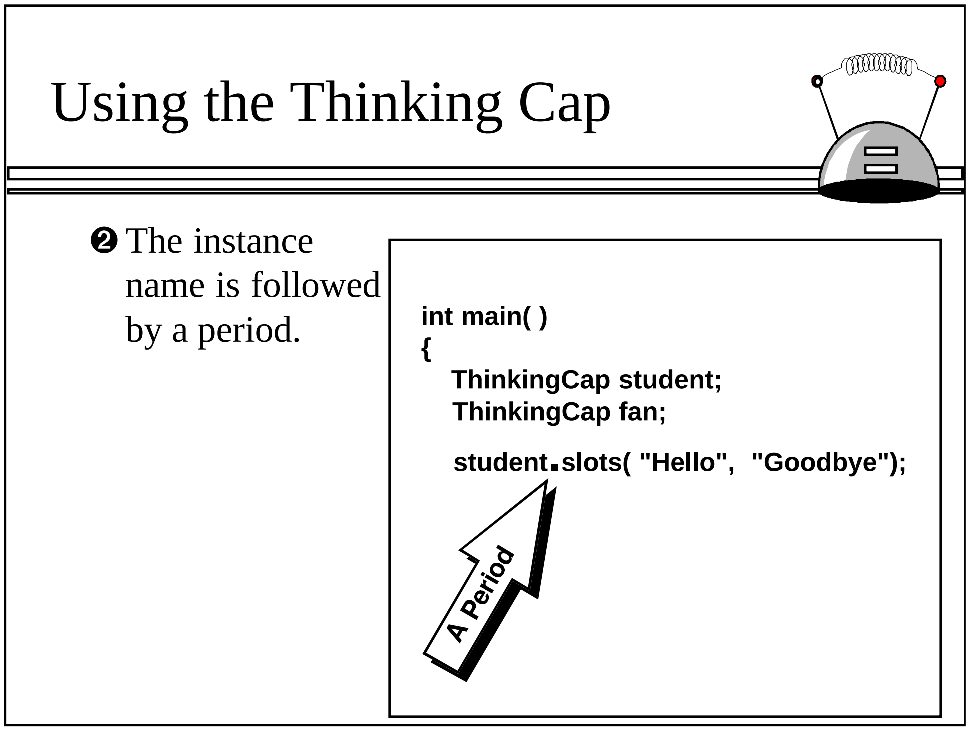
Using the Thinking Cap

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

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

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

Name of the object

The complete activation consists of four parts, beginning with the object name.



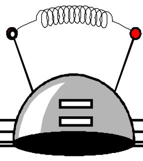
Using the Thinking Cap

❷ The instance name is followed by a period.

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

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

The object name is followed by a period.

Using the Thinking Cap

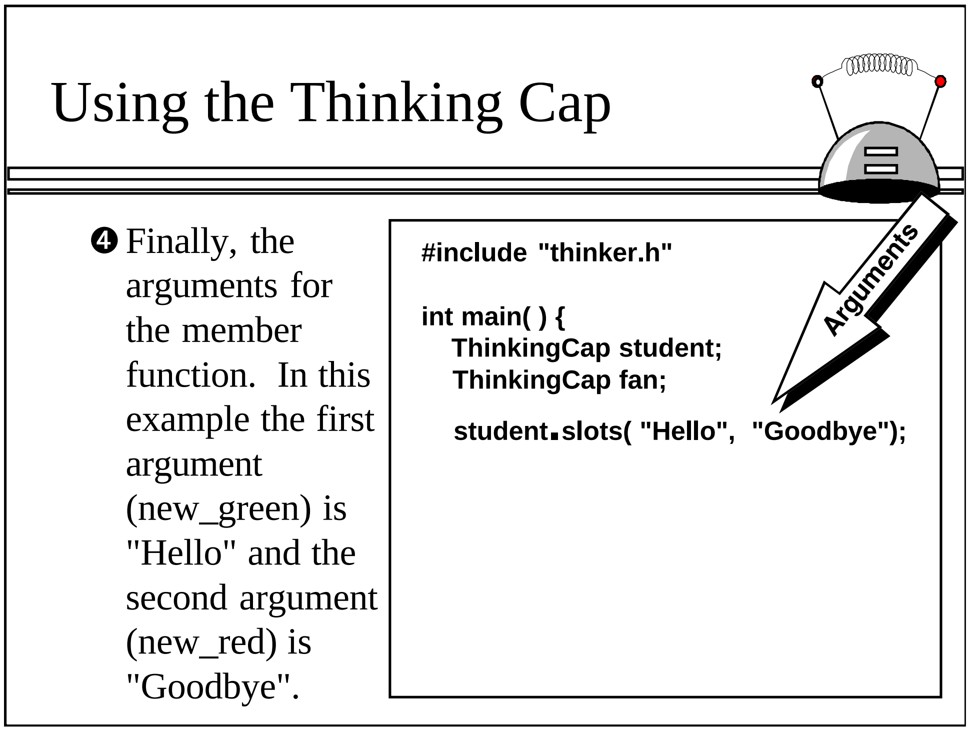


❸ After the period is the name of the member function that you are activating.

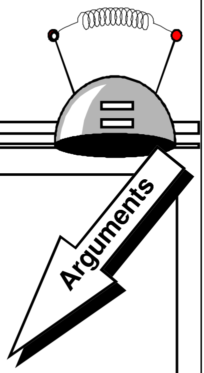
```
int main( ) {
    ThinkingCap student;
    ThinkingCap fan;

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

Name of the Function

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



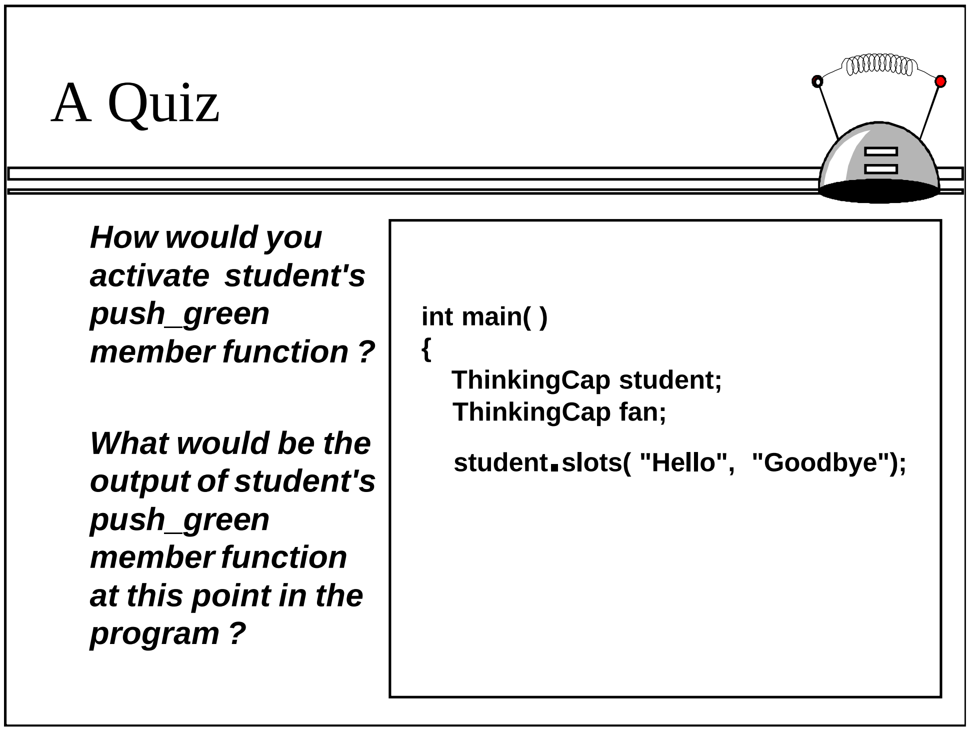
Using the Thinking Cap



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

```
#include "thinker.h"

int main( ) {
   ThinkingCap student;
   ThinkingCap fan;

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

And finally there is the argument list. In the case of the slots member function, there are two string arguments: new_green (which is given the actual value "Hello" in this example) and new_red (which is given the actual value "Goodbye" in this example).



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",  "Goodbye");
```

Go ahead and write your answers before I move to the next slide.

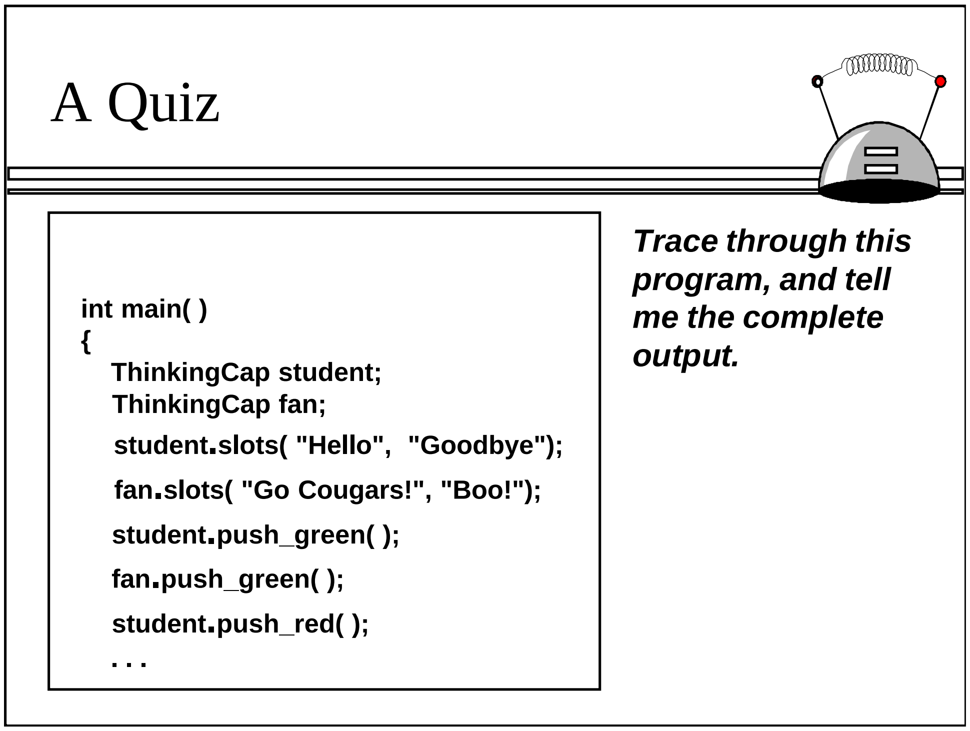
A Quiz

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",  "Goodbye");

    student.push_green( );
```

Remember that the push_green member function does not have any arguments, so the argument list is just a pair of parentheses.

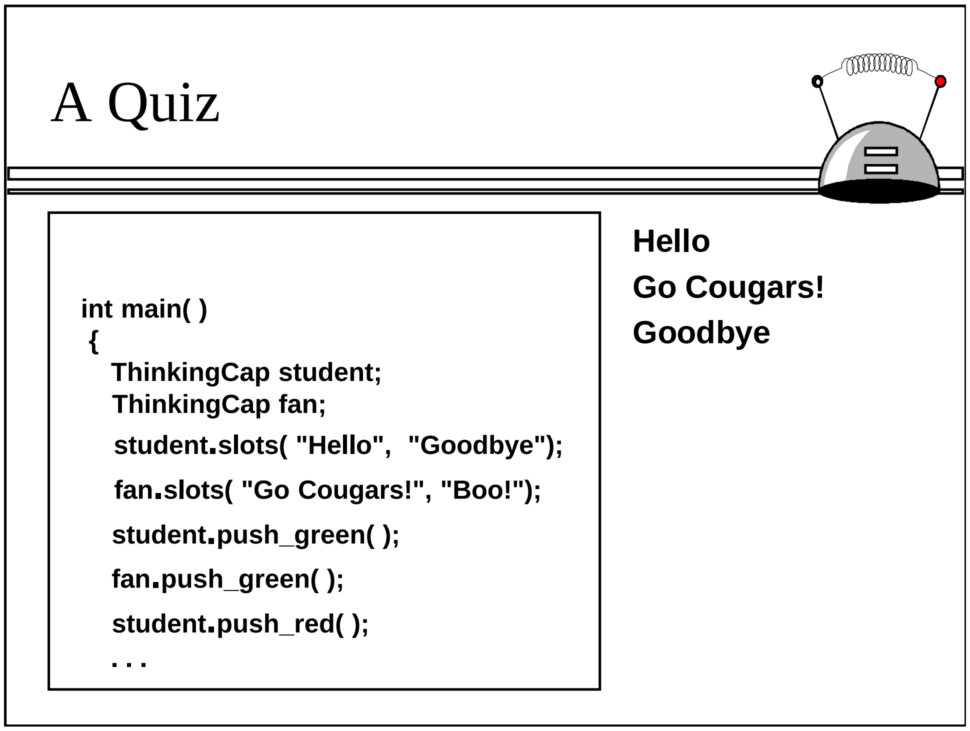


A Quiz

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

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

Here's a longer program. What is the complete output? Again, write your answers before I move to the next slide.



A Quiz

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

```
Hello
Go Cougars!
Goodbye
```

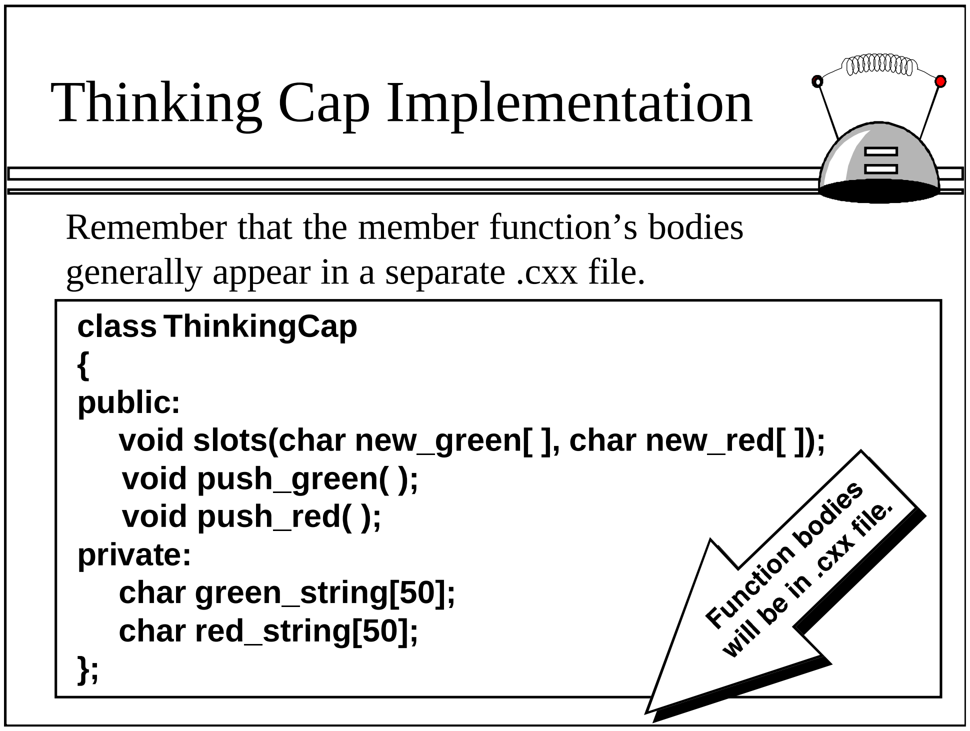
The important thing to notice is that student and fan are separate objects of the ThinkingCap class. Each has its own green_string and red_string data. Or to throw one more piece of jargon at you: Each has its own green_string and red_string member variables. Member variables are the data portion of a class. The activation of student.slots fills in the green_string and red_string data for student, and the activation of fan.slots fills in the green_string and red_string data for fan.

Once these member variables are filled in, we can activate the push_green and push_red member functions. For example, student.push_green accesses the green_string member variable of student, whereas fan.push_green accesses the green_string member variable of fan.

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.

You now know quite a bit about OOP -- but the key missing piece is how to implement a class’s member functions.



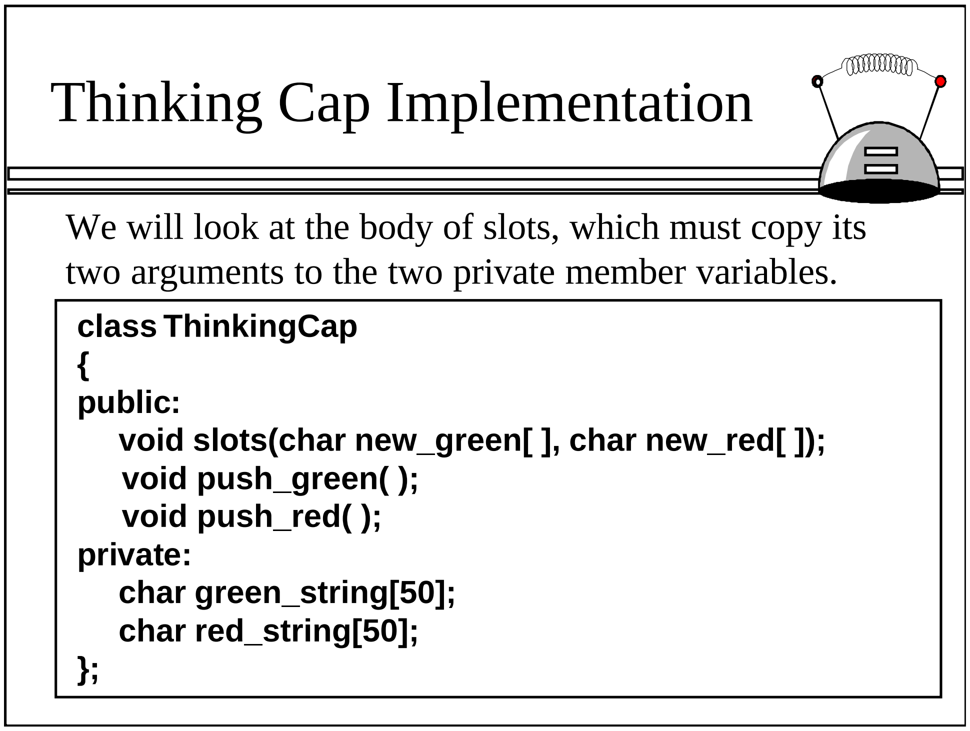
Thinking Cap Implementation

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

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

Function bodies
will be in .cxx file.

You already know the location of these implementations: in a separate “implementation file” called thinker.cxx.

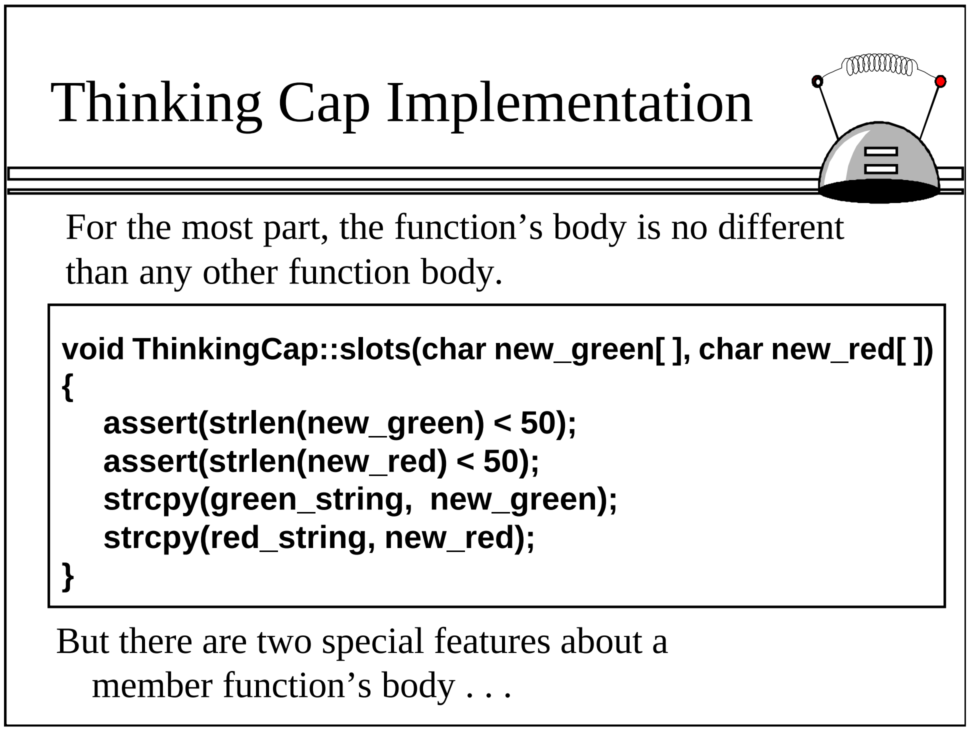


Thinking Cap Implementation

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( );
    void push_red( );
private:
    char green_string[50];
    char red_string[50];
};
```

We'll start by looking at the implementation of the slots member function. The work which the function must accomplish is small: Copy the two arguments (new_green and new_red) to the two private member variables of the object (green_string and red_string).



Thinking Cap Implementation

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

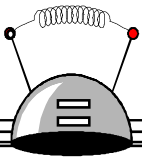
```
void ThinkingCap::slots(char new_green[ ], char new_red[ ])
{
    assert(strlen(new_green) < 50);
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_green);
    strcpy(red_string, new_red);
}
```

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

For the most part, all that's needed is a pair of calls to strcpy to copy the two arguments (new_green and new_red) to the two member variables (green_string and red_string). By the way, how many of you have seen this use of strcpy before? If you haven’t seen it, don’t worry--it will be covered in Chapter 4. For now all you need to know is that the strcpy statements work like assignment statements, copying from the right to the left. Also, you might notice that we have checked to make sure that the new_green and new_red have fewer than 50 characters (using strlen, which returns the number of characters in a string).

But the more interesting parts of this implementation are two special features that you need to know about.

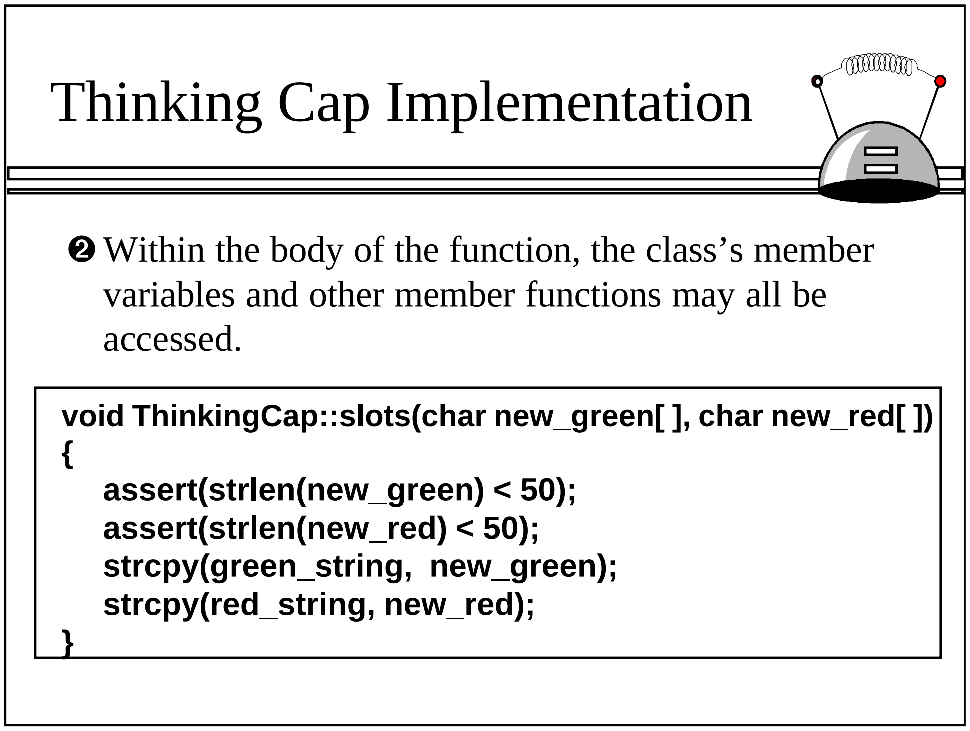
Thinking Cap Implementation



❶ 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) < 50);
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_green);
    strcpy(red_string, new_red);
}
```

First of all, in the member function's heading you must include the name of the class followed by two colons, as shown here. Otherwise, the C++ compiler will think that this is an ordinary function called slots, rather than a member function of the ThinkingCap class.

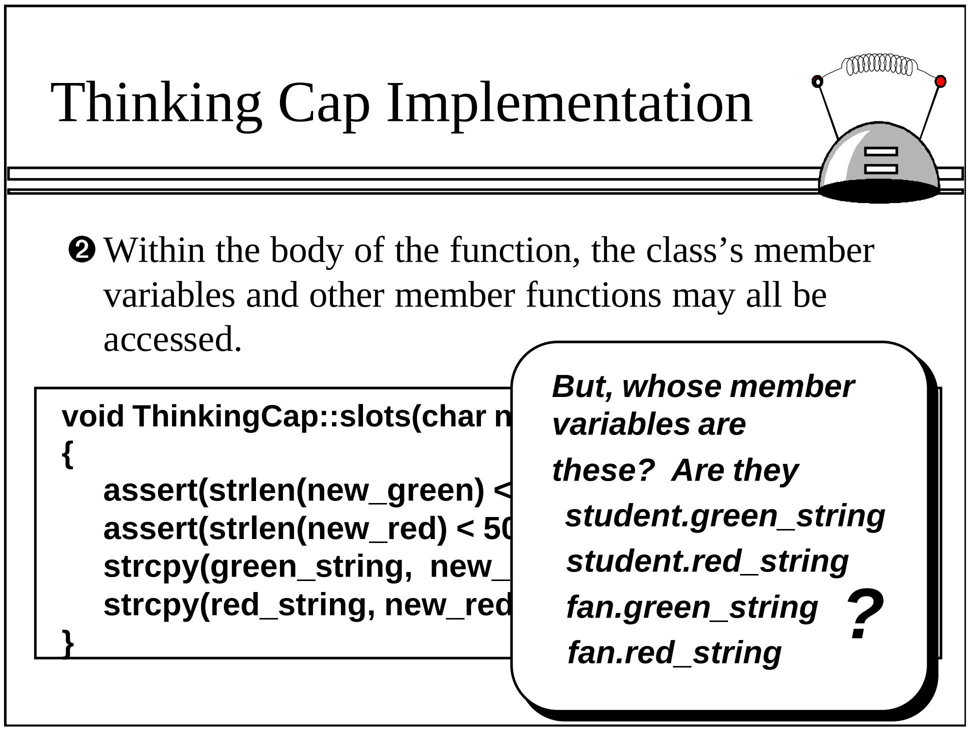


Thinking Cap Implementation

❷ 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) < 50);
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_green);
    strcpy(red_string, new_red);
}
```

Within the body of the member function, we may access any of the members of the object. In this example, we are accessing both the green_string and the red_string member variables, by assigning values to these member variables.



Thinking Cap Implementation

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

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

But, whose member variables are these? Are they

student.green_string

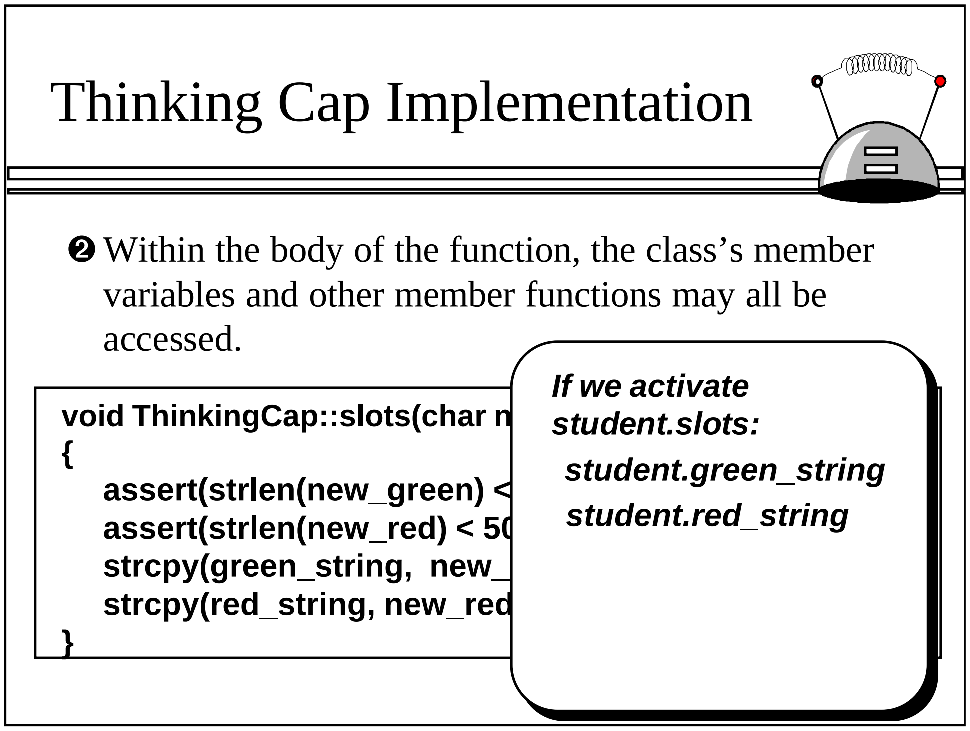
student.red_string

fan.green_string

fan.red_string

?

The use of these member variables is a bit confusing. Which member variables are we talking about? student.green_string and student.red_string? Or are we referring to fan.green_string and fan.red_string? Or member variables of some other object?



Thinking Cap Implementation

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

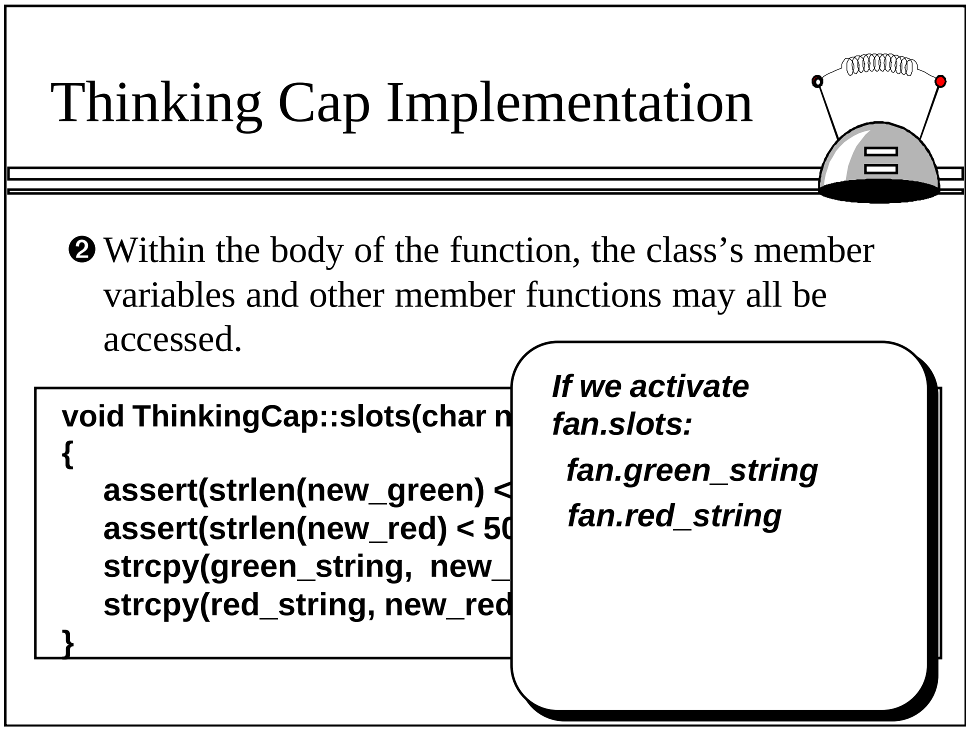
```
void ThinkingCap::slots(char n
{
    assert(strlen(new_green) <
    assert(strlen(new_red) < 50
    strcpy(green_string,  new_
    strcpy(red_string, new_red
}
```

If we activate student.slots:

student.green_string

student.red_string

The answer depends on which object has activated its member function. If student.slots is activated, then these two member variables will refer to student.green_string and student.red_string.



Thinking Cap Implementation

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

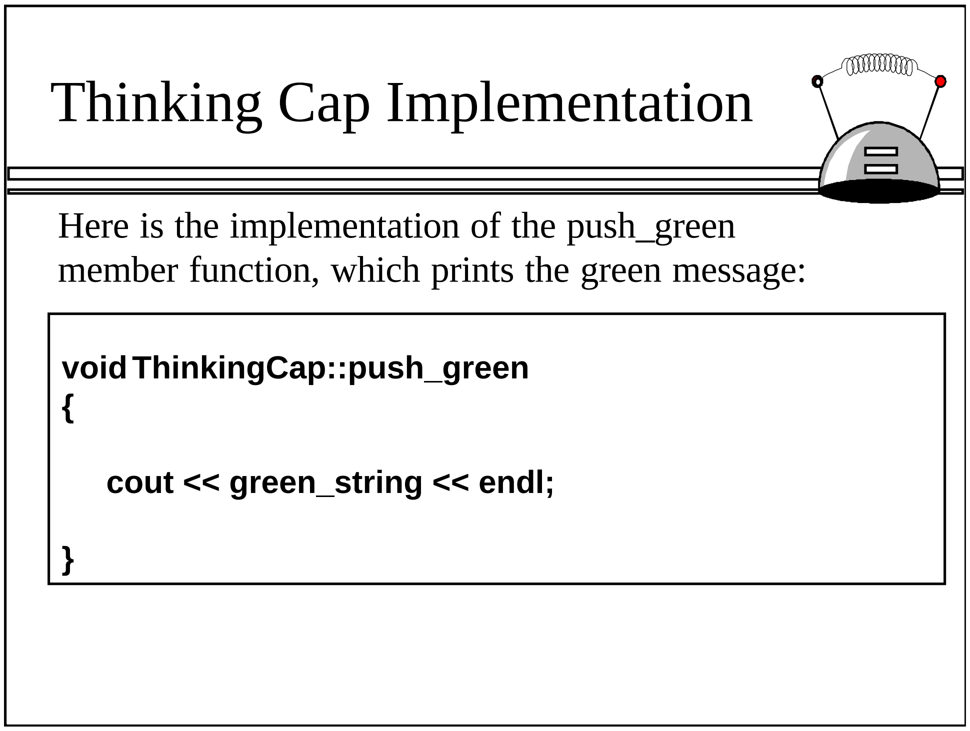
```
void ThinkingCap::slots(char n
{
    assert(strlen(new_green) <
    assert(strlen(new_red) < 50
    strcpy(green_string,  new_
    strcpy(red_string, new_red
}
```

If we activate fan.slots:

fan.green_string

fan.red_string

But if fan.slots is activated, then the two member variables refer to fan.green_string and fan.red_string.

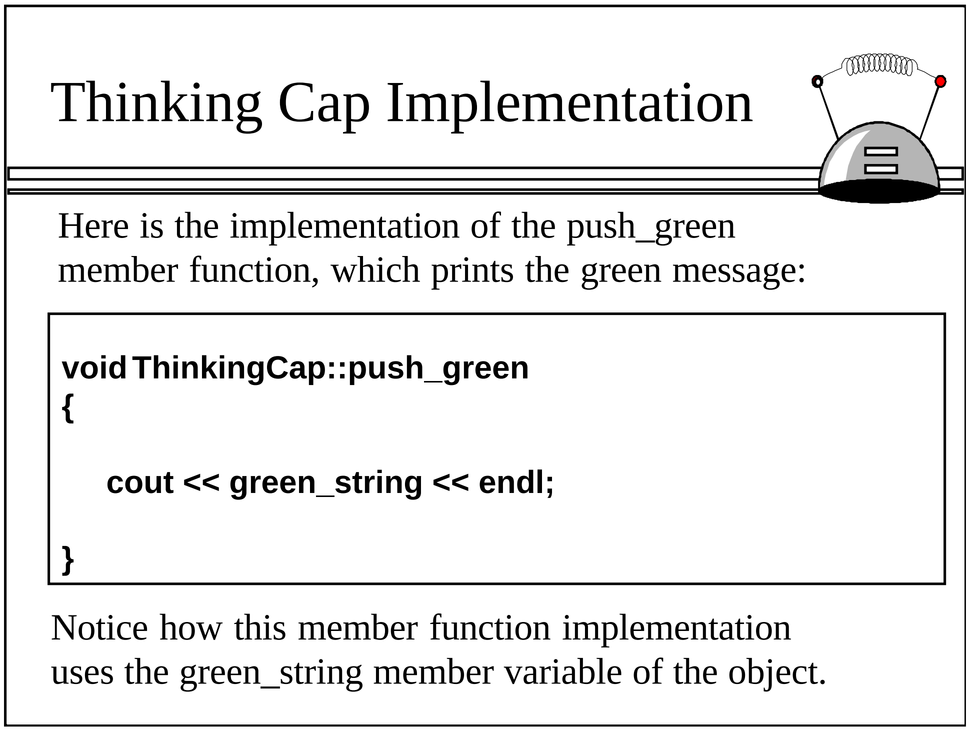


Thinking Cap Implementation

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

```
void ThinkingCap::push_green
{

    cout << green_string << endl;

}
```

Here's the implementation of the push_green member function.



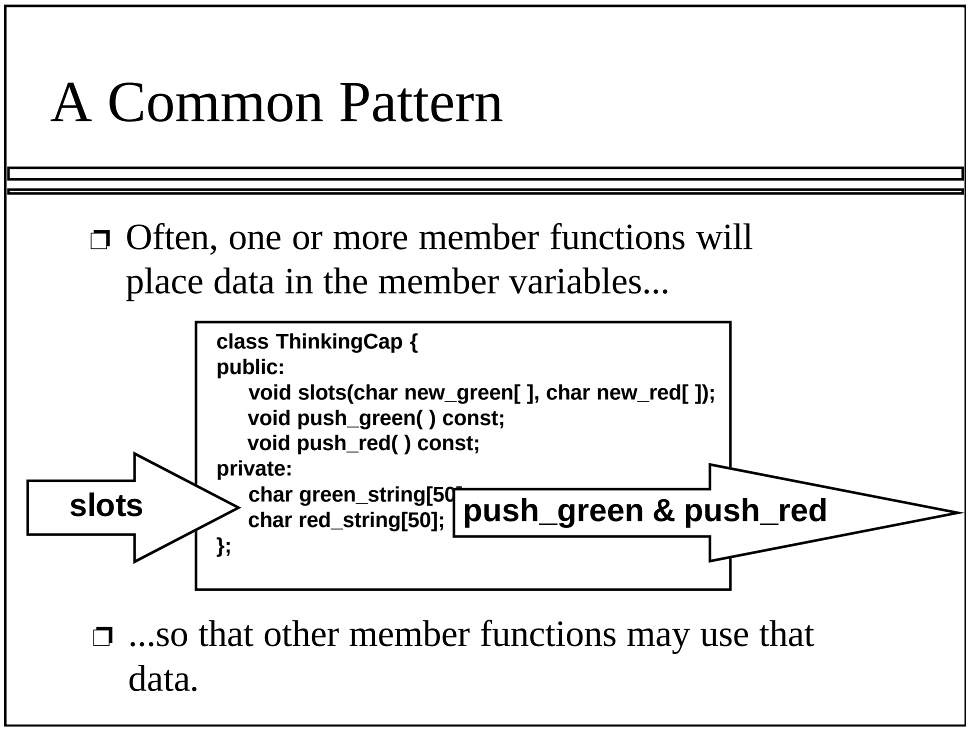
Thinking Cap Implementation

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

```
void ThinkingCap::push_green
{

    cout << green_string << endl;

}
```

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

The important thing to notice is how the member function's implementation uses the green_string member variable of the object. If we activate student.push_green, then the member function will use student.green_string. And if we activate fan.push_green, then the member function will use fan.green_string.



A Common Pattern

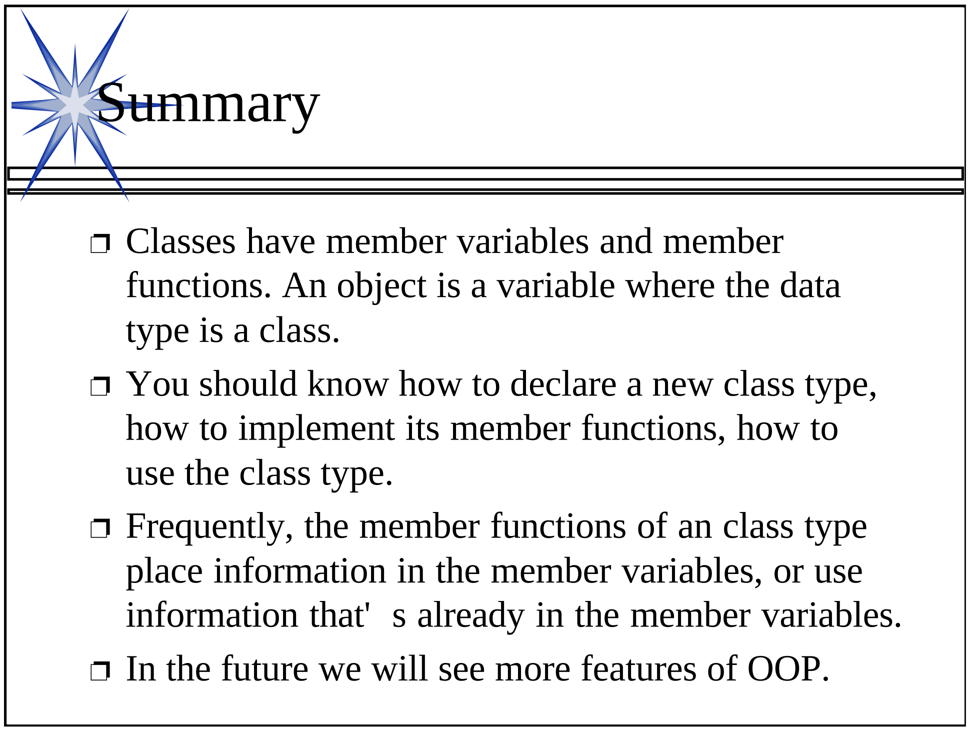
- ❒ Often, one or more member functions will place data in the member variables...

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

slots
push_green & push_red

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

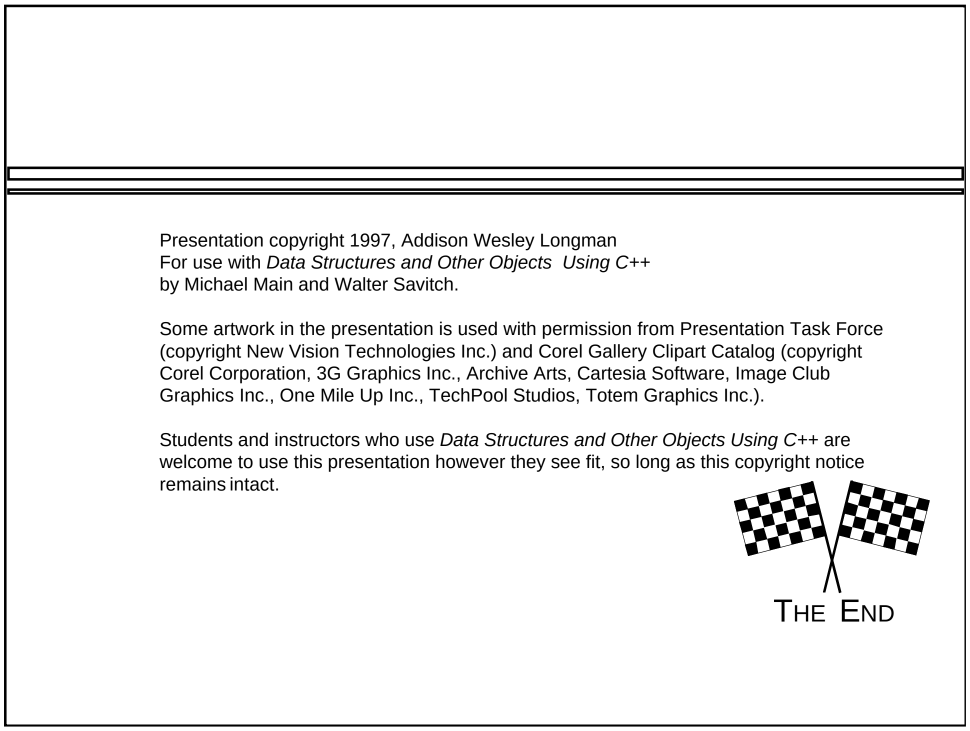
The member functions of the ThinkingCap are all simple, but they do illustrate a common pattern: Often a member function (such as slots) will place information in the private member variables, so that other const member functions (such as push_green and push_red) may access the information in those member variables.



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.

A quick summary . . . This presentation has only introduced classes. You should read all of Chapter 2 to get a better understanding of classes. Pay particular attention to the notion of a constructor, which is a special member function that can automatically initialize the member variables of an object. Also pay attention to the more advanced features such as operator overloading with the Point class given in Chapter 2.



THE END