C++ CODE DESIGN
INTRO TO ABSTRACT DATA TYPES

Problem Solving with Computers-II

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

int main(){
    cout<<"Hola Facebook\n";
    return 0;

}
```

https://ucsb-cs24-sp17.github.io/

Read the syllabus. Know what's required. Know how to get help.

CLICKERS OUT – FREQUENCY AB

Intro to specification and design

- Chapter 1 introduces the software development cycle
- Key concepts: Specification, design, implementation

(What is your understanding of what each of these are – discuss)

Example

- You are the head of a programming team and you want one of your programmers to write a function for part of a project.

HERE ARE THE REQUIREMENTS FOR A FUNCTION THAT I WANT YOU TO WRITE.
I DON'T CARE WHAT METHOD THE FUNCTION USES, AS LONG AS THESE REQUIREMENTS ARE MET.

Procedural abstraction

Frequently a programmer must communicate precisely **what** a function accomplishes, without any indication of **how** the function does its work.

Specifying function behavior

Specifying behaviour of a function with information hiding?

- The **precondition** statement indicates what must be true before the function is called.
- The **postcondition** statement indicates what will be true when the function finishes its work.

Example

```
void write_sqrt(double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.




    • • •
```

Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.




}
```

- The precondition and postcondition appear as comments in your program.

Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
}
```

- In this example, the precondition requires that

  x >= 0

  be true whenever the function is called.

Example

Which of these function calls does not meet the precondition ?

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
}
```

- The postcondition always indicates what work the function has accomplished. In this case, when the function returns the square root of **x** has been written.

Another Example

```
bool is_vowel( char letter )
//   Precondition:  letter is an uppercase or
//   lowercase letter (in the range 'A' ... 'Z' or 'a' ... 'z') .
//   Postcondition:  The value returned by the
//   function is true if Letter is a vowel;
//   otherwise the value returned by the function is
//   false.



   • • •
```

Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel(' Z' );
is_vowel( '?' );
```

Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel(' Z' );
is_vowel( '?' );
```

true

false

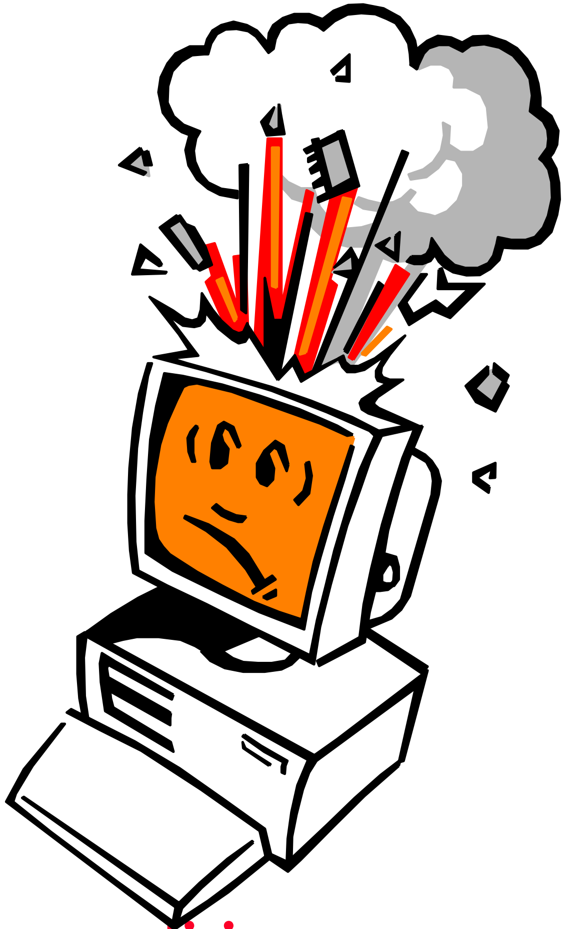
Nobody knows, because the precondition has been violated.

Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel(' Z' );
is_vowel( '?' );
```

Violating the precondition
might even crash the entire program.



A Quiz

Suppose that you call a function, and you neglect to make sure that the precondition is valid. Who is responsible if this inadvertently causes a 40-day flood or other disaster?

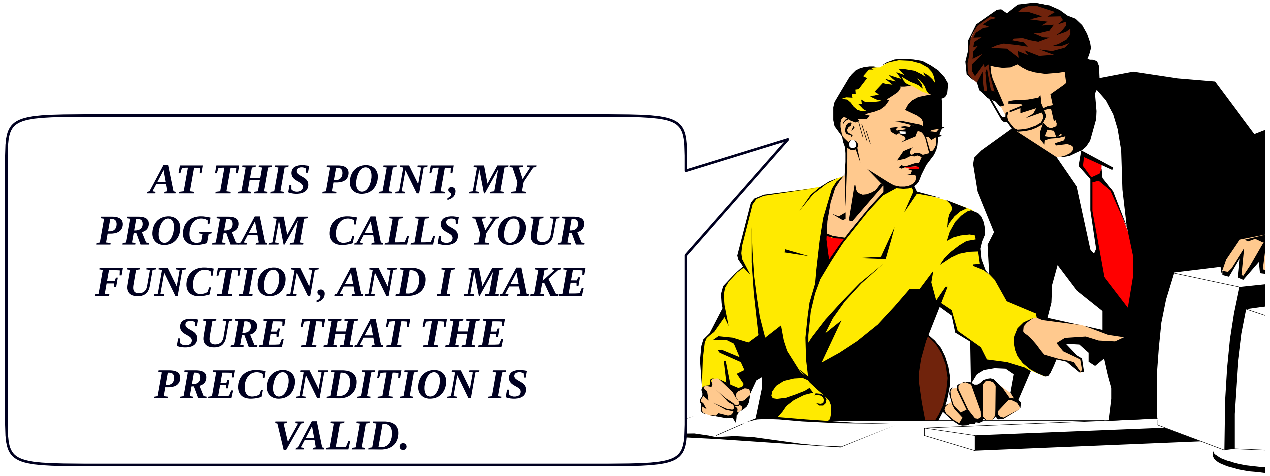
A. You

B. The programmer who wrote that torrential function

C. Noah

Always make sure the precondition is valid . . .

- The programmer who calls the function is responsible for **ensuring that the precondition is valid** when the function is called.


AT THIS POINT, MY PROGRAM CALLS YOUR FUNCTION, AND I MAKE SURE THAT THE PRECONDITION IS VALID.

. . . so the postcondition becomes true at the function’s end.

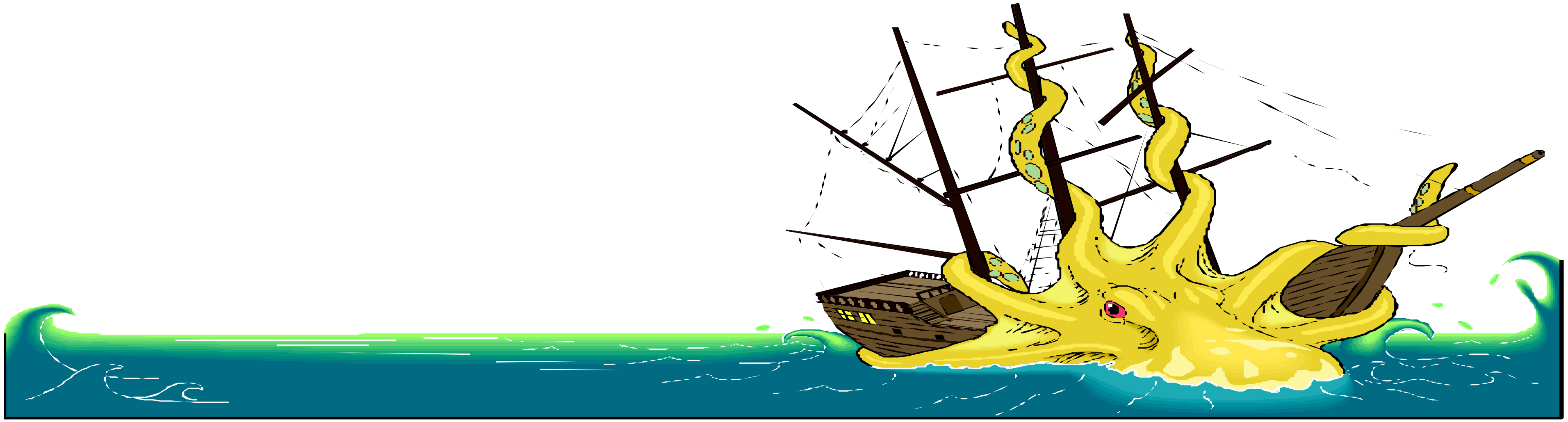
- The programmer who writes the function counts on the precondition being valid, and **ensures that the postcondition becomes true** at the function’s end.

THEN MY FUNCTION WILL EXECUTE, AND WHEN IT IS DONE, THE POSTCONDITION WILL BE TRUE.
I GUARANTEE IT.

On the other hand, careful programmers also follow these rules:

- Detect when a precondition has been violated.
- Print an error message and halt the program...

  ...rather than causing a disaster.



Which of the following statements would you use to detect if a precondition has been violated?

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{


  ____________


  //Program implementation
}
```

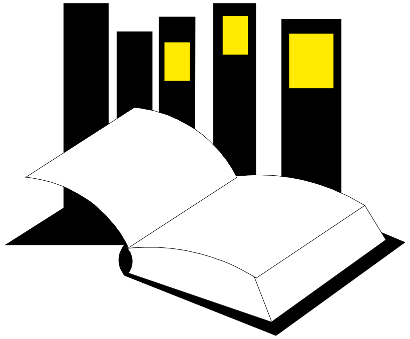
A. if(x<0) return;
B. assert(x >= 0);
C. if(x<0) cerr<<“Input “<<x<< “ is less than 0”;
D. Option B or C
E. Any of the above would work

Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{
    assert(x >= 0);

  • • •
```

The assert function (described in Section 1.1) is useful for detecting violations of a precondition.

Intro to Object Oriented Programming

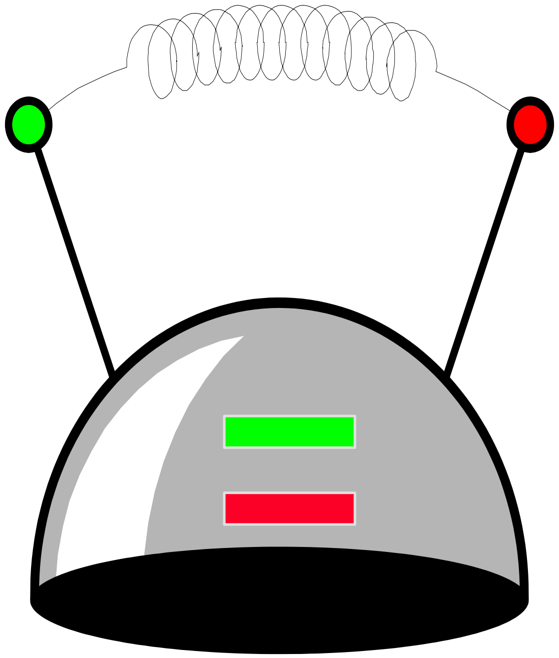


Data Structures and Other Objects Using C++

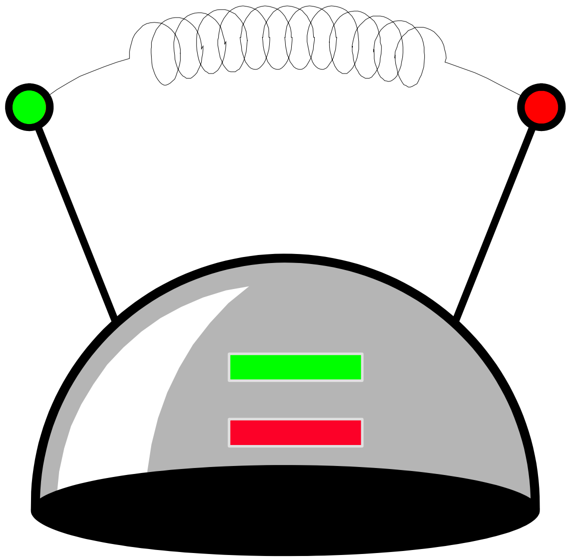
- Chapter 2 introduces Object Oriented Programming.
- OOP is an approach to programming which supports the creation of new data types and operations to manipulate those types.

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.

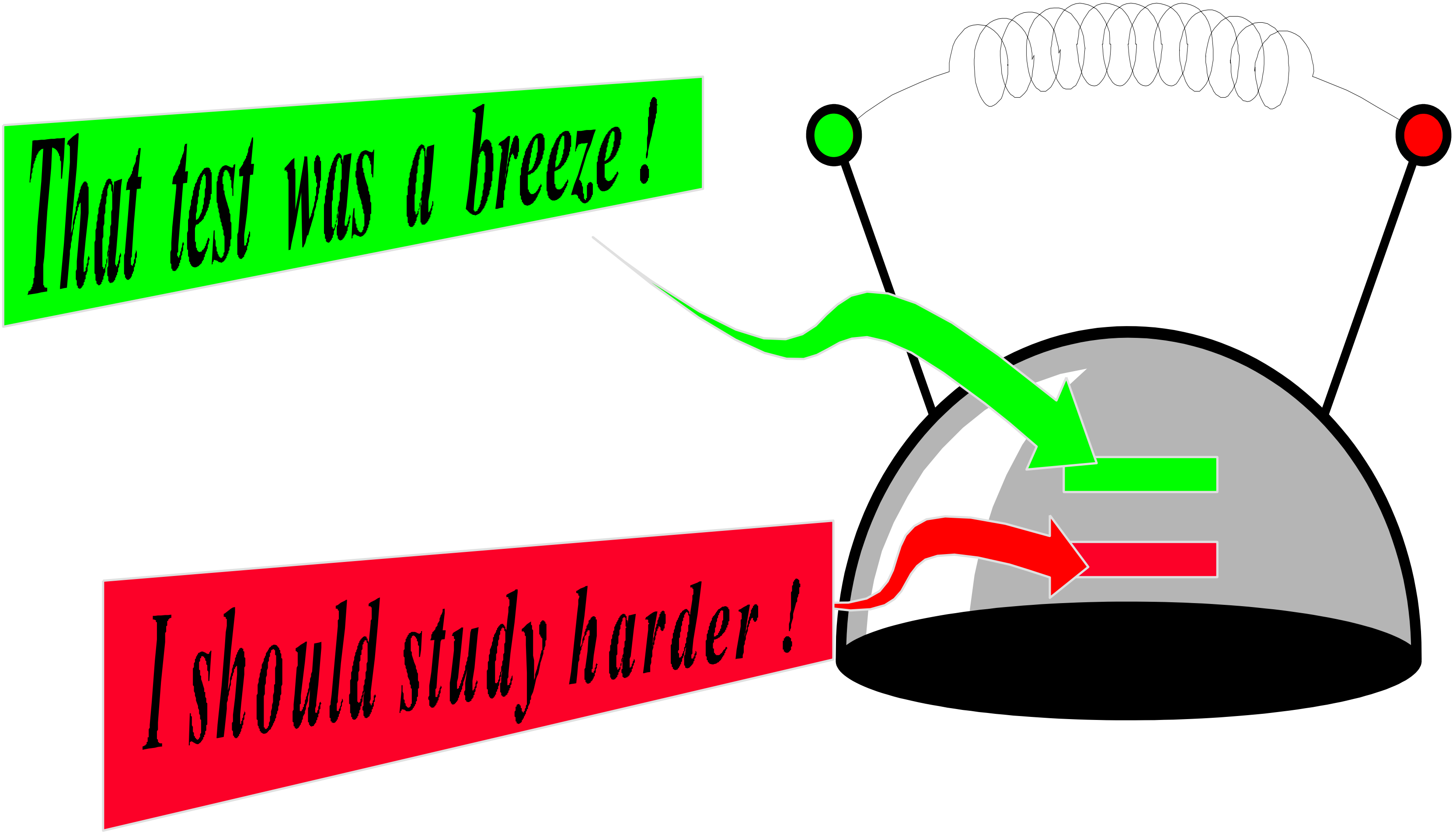


Description of the thinking cap



- 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.

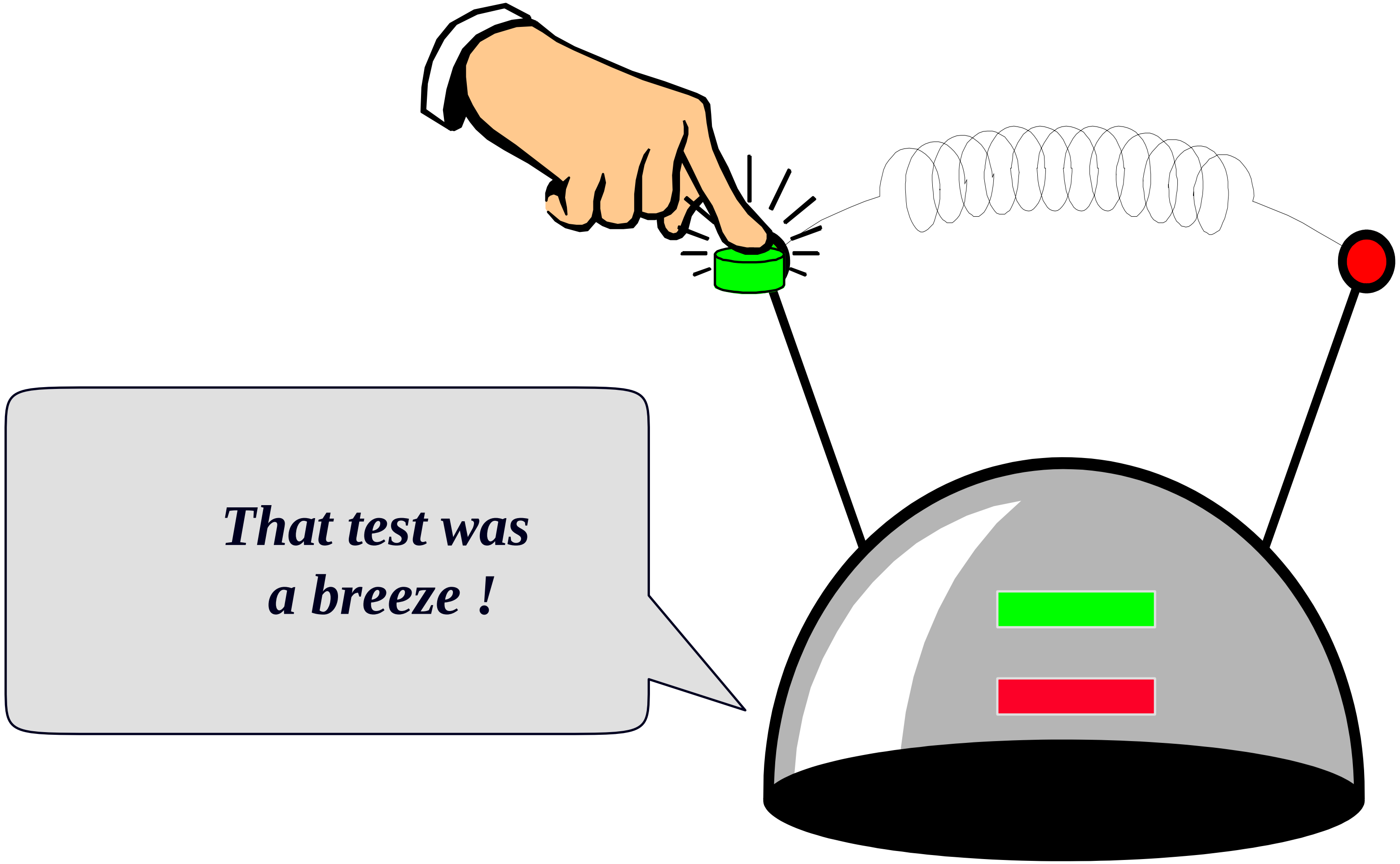
Example



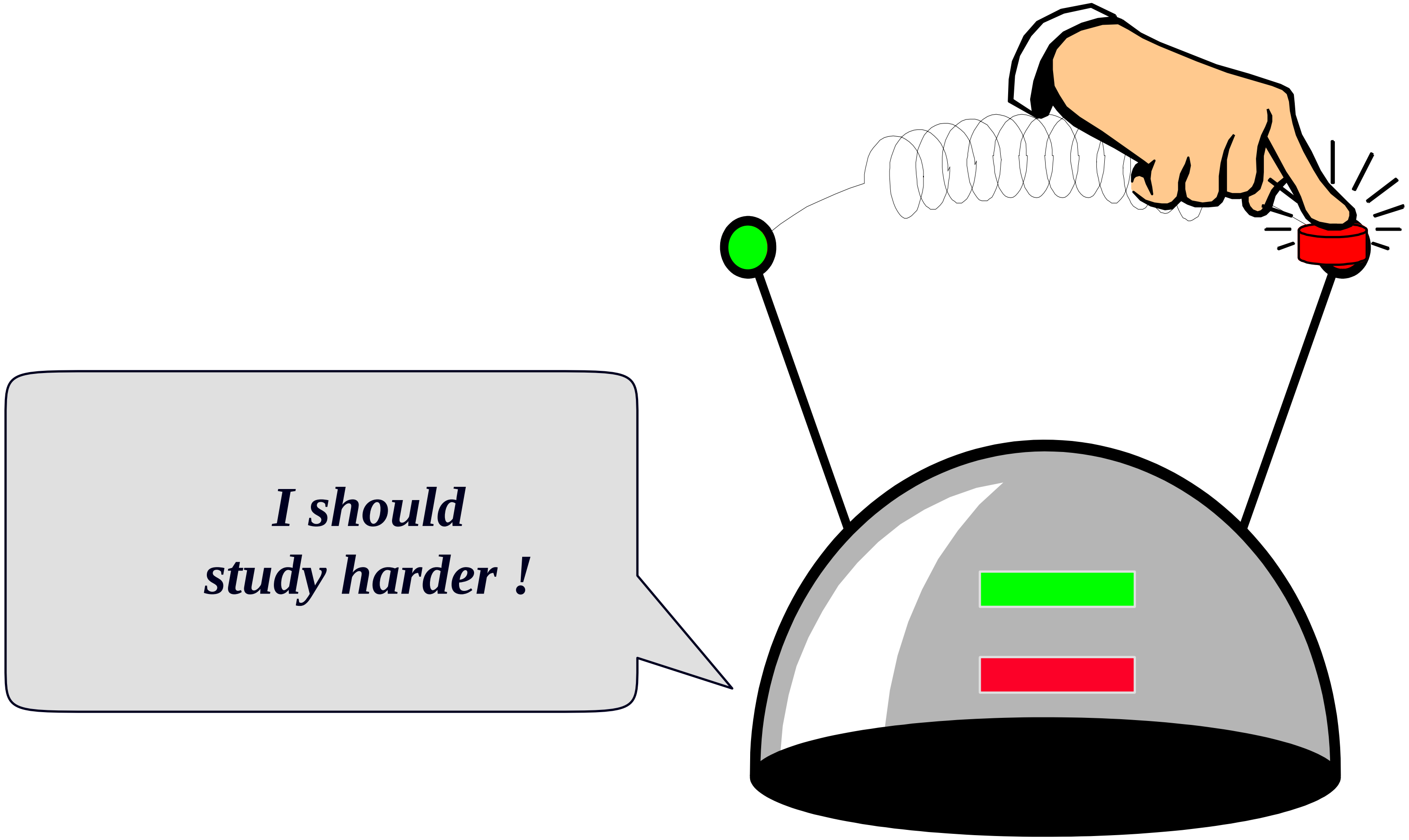
That test was a breeze !

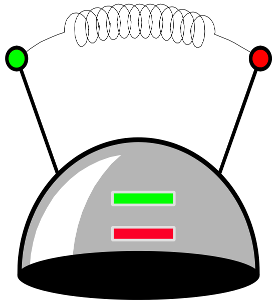
I should study harder !

Example



Example



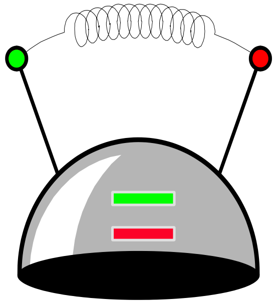


Thinking Cap Definition

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

```
class thinking_cap
{

          . . .

 };
```

Components of the thinking cap

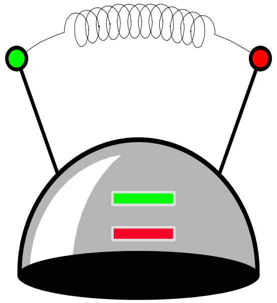


- The class will have two components called green_string and red_string..

How is a class different from a struct?

```
class thinking_cap
{

    char green_string[50];
    char red_string[50];
    . . .
};
```

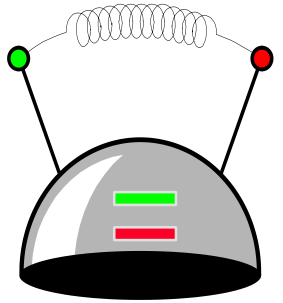
Thinking Cap as an Abstract Data Type



- 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 thinking_cap
{

private:
    char green_string[50];
    char red_string[50];
};
```

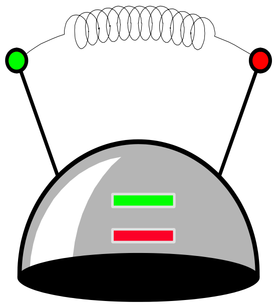
Thinking Cap as an Abstract Data Type



- Public interface – can be accessed by the user of the class
  - List member function (methods) that manipulate data here!
  - Provide a clear interface to data!!

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

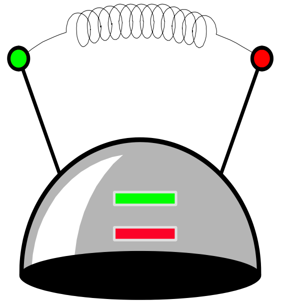
Thinking Cap Implementation



- List member function (methods) that manipulate data – ONLY declarations

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

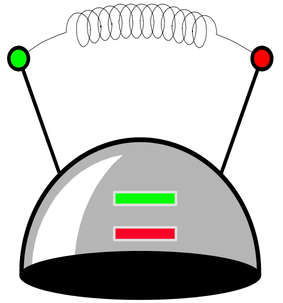
Thinking Cap Implementation



```
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];
};
```

Function bodies will be elsewhere.

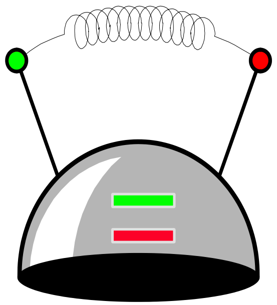
Thinking Cap Implementation



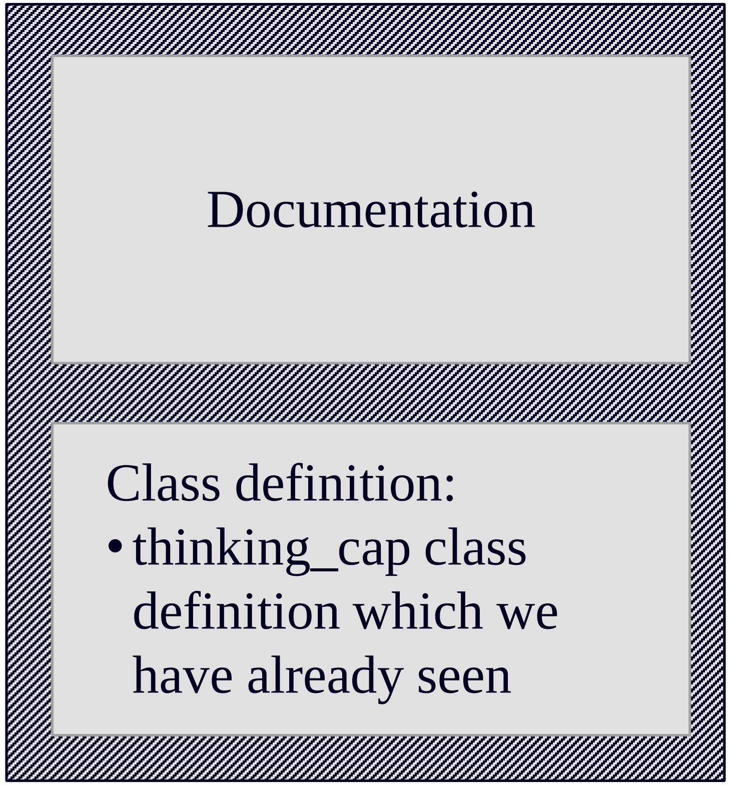
```
class thinking_cap
{
public:
    void slots(char new_green[ ], char          ed[ ]);
    void push_green( ) const;
    void push_red( ) const;
private:
    char green_string[50];
    char red_string[50];
};
```

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

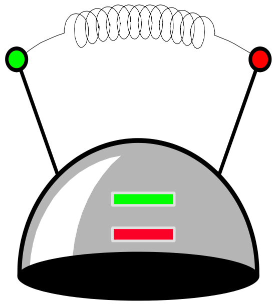


Documentation

Class definition:

- thinking_cap class definition which we have already seen

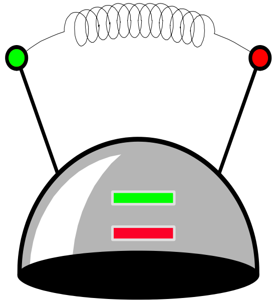
Using the Thinking Cap



- 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"

...
```

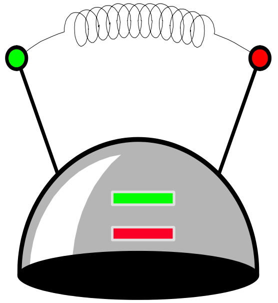
Using the Thinking Cap



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

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

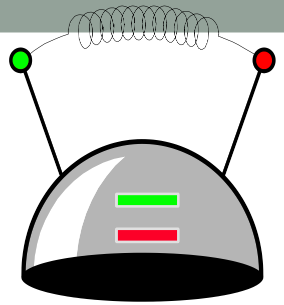
- How is student different from “thinking_cap”?
- What happens in memory after this code is executed?

Using the Thinking Cap



- Activating the student’s slot method

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

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



A Quiz

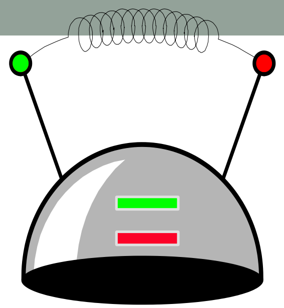
How would you activate student's push_green member function ?

(Write your answer)

(After that discuss with your peer group)

```
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];
};
```

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



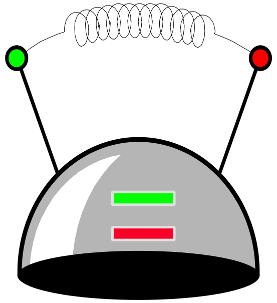
A Quiz

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

```
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];
};

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 Cougars!", "Boo!");
    student.push_green( );
    fan.push_green( );
    student.push_red( );
```



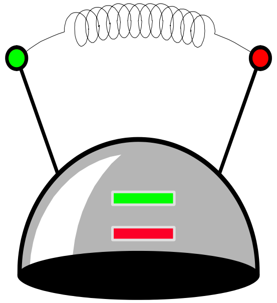
Trace through this program, and tell me the complete output.

What you know so far?

- Class = Data + Member Functions.
- Abstract Data Type = Class + information hiding
- 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 methods.

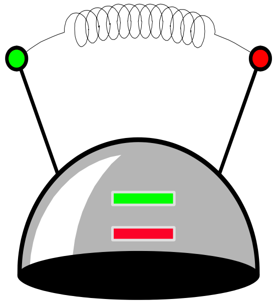
Thinking Cap Implementation



```
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];
};
```

Function bodies will be in .cxx file.

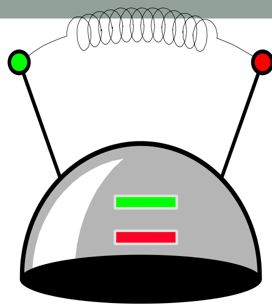
Thinking Cap Implementation



- Implement the class in a separate .cxx file.
- With your peer group implement the slots function

```
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];
};
```

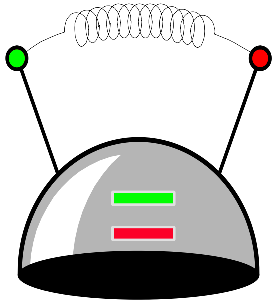
Thinking Cap Implementation



There are two special features about a member function's 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);
}
```

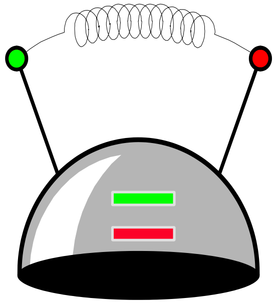
Thinking Cap Implementation



- Why use the scope resilution operator?

```
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 methods may all be accessed.

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

But, whose member variables are

these? Are they

student.green_string

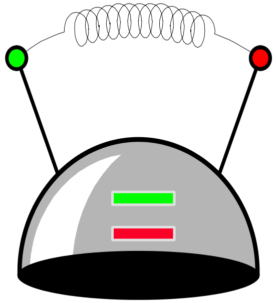
student.red_string

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.

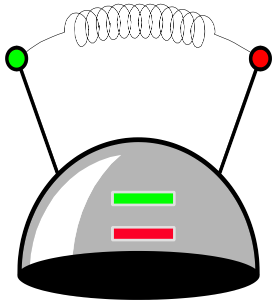
```
void thinking_cap::slots(char new_
{
    assert(strlen(new_green) < 50)
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_gree
    strcpy(red_string, new_red);
}
```

If we activate student.slots:

student.green_string

student.red_string

Thinking Cap Implementation



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

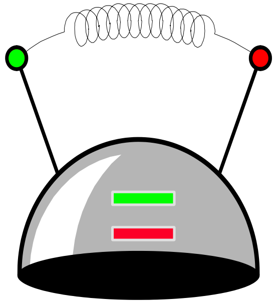
```
void thinking_cap::slots(char new_
{
    assert(strlen(new_green) < 50)
    assert(strlen(new_red) < 50);
    strcpy(green_string,  new_gree
    strcpy(red_string, new_red);
}
```

If we activate fan.slots:

fan.green_string

fan.red_string

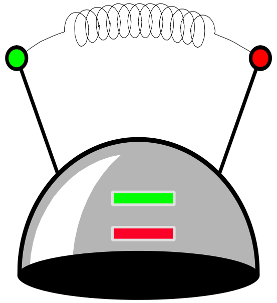
Thinking Cap Implementation



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

```
void thinking_cap::push_green
{

    cout << green_string << endl;

}
```

Thinking Cap Implementation



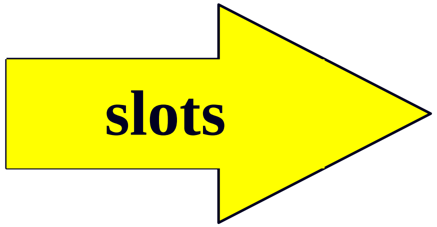
Here is the implementation of the push_green member function, which prints the green message:

```
void thinking_cap::push_green
{

     cout << green_string << endl;

}
```

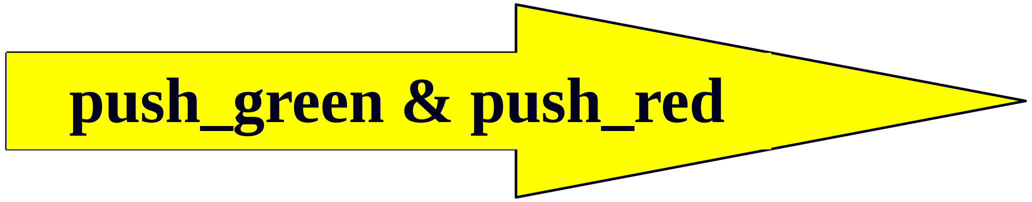
A Common Pattern

- 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];
};
```



slots



push_green & push_red

Summary

- Classes have member variables and member functions (method). 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.

Next time

- Constructors