

# Abstraction and OOP

# Today's Plan



Announcements

Recap

Abstraction

OOP

# Announcements

- If you already have a Gradescope account from previous semesters you still need to make sure you have access to this course
- **Project 1 posted today!**

# Recap

Minimize software size and interactions

Simplify complex program to manageable level

Break down into smaller problems

Isolate functionalities

Minimize and control interactions

So how do we do this?

# Abstraction

# Abstraction Example



# Abstraction Example



You always use them,  
switch from one to another  
seamlessly and probably  
don't think too much  
about them



# Printers

Come in all shapes and sizes

Can have different complex mechanisms  
(Laser, Laserjet, Inkjet, Dot matrix ... )

**Easy to use**

**- something common to all of them - abstraction**



# What is a printer?

# What is a printer?

A printer reproduces graphics or text on paper

# What is a printer?

A printer reproduces graphics or text on paper

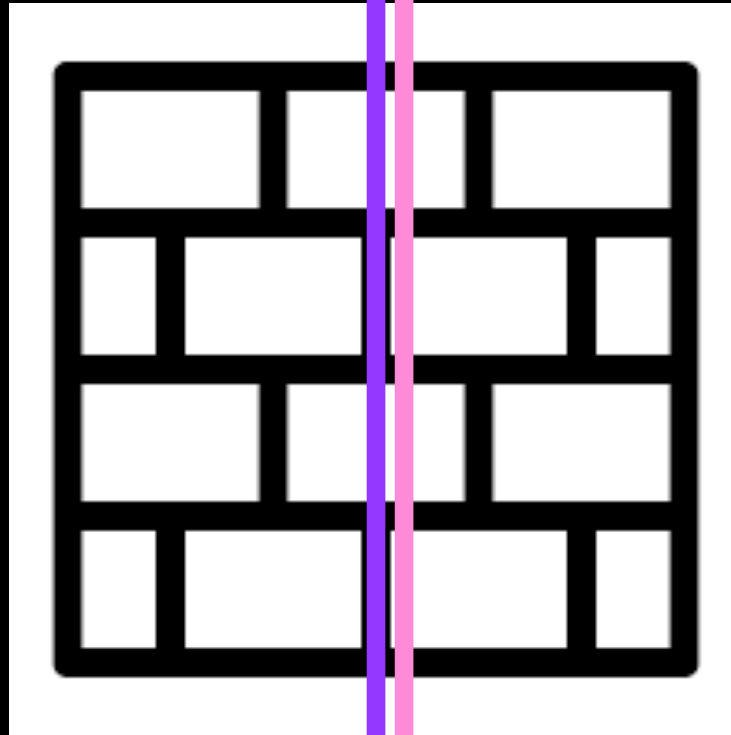
Separate functionality from implementation  
(i.e. what can be done from how it's actually done)

# Wall of Abstraction

Information barrier between device (program) use and how it works

Painstaking work to  
design technology  
and implement  
printers

**Design and  
implementation**



Press button  
Or  
Send print job from  
application

**Usage**

# Abstractions are imprecise

A printer reproduces graphics or text on paper

Wall of abstraction between *implementer* and *client*

How does client know how to use it?

# Abstractions are imprecise

A printer reproduces graphics or text on paper

Wall of abstraction between *implementer* and *client*


How does client know how to use it?

Provide an *interface* (what the user needs to interact)

In Software Engineering typically a set of *attributes* (data or properties) and a set of *actions*

# Lecture Activity

**Attributes (data):**

A yellow speech bubble with a black border and a tail pointing towards the left. It contains text about designing the interface.

**Designing the interface:**  
think about what the user needs  
to do / know about

**Actions (operations):**

# Interface for Printer

## Attributes (data):

- Ink level
- Paper level
- Error codes

## Actions (operations):

- Print
- Rotate (landscape/portrait)
- Color / Black & White



How this is done  
is irrelevant to  
the client



# Information Hiding

In this course  
it always means software

Interface —> **client** doesn't have to know about the inner workings

Actually client **shouldn't** know of or *have* access to implementation details

It is **dangerous** to allow clients to bypass interface



Safe Programming

# Reasons for Information Hiding

**Harmful** for client to tamper with someone else's implementation (*code*)

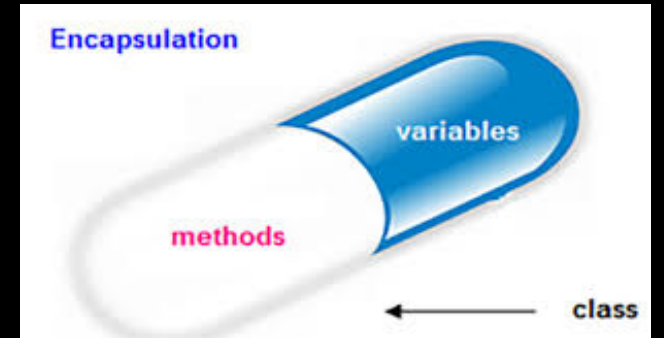
- Voluntarily/involuntarily **break it - misuse it**
- **Reduces flexibility and modifiability** by locking implementation in place
- Increases number of **interactions** between modules

# Object Oriented Design

# Principles of Object Oriented Programming (OOP)

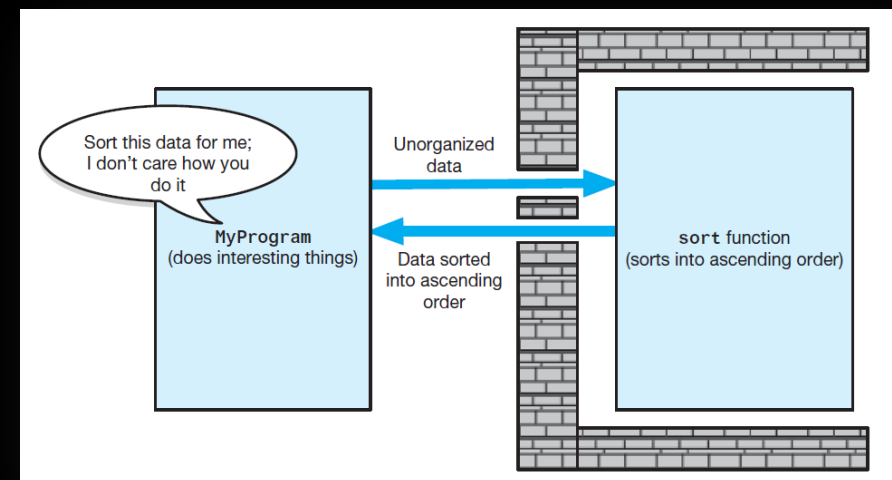
## Encapsulation

*Objects combine data and operations*



## Information Hiding

*Objects hide inner details*



## Inheritance

*Objects inherit properties from other objects*

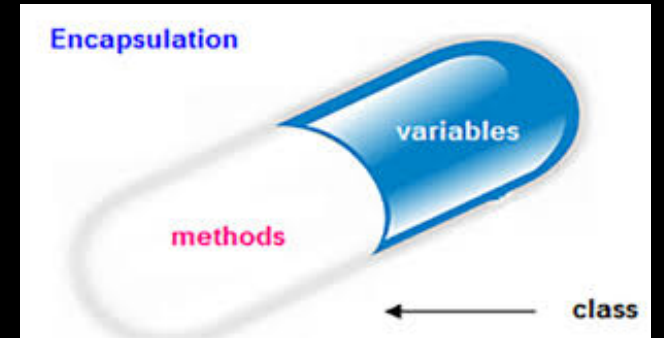
## Polymorphism

*Objects determine appropriate operations at execution*

# Principles of Object Oriented Programming (OOP)

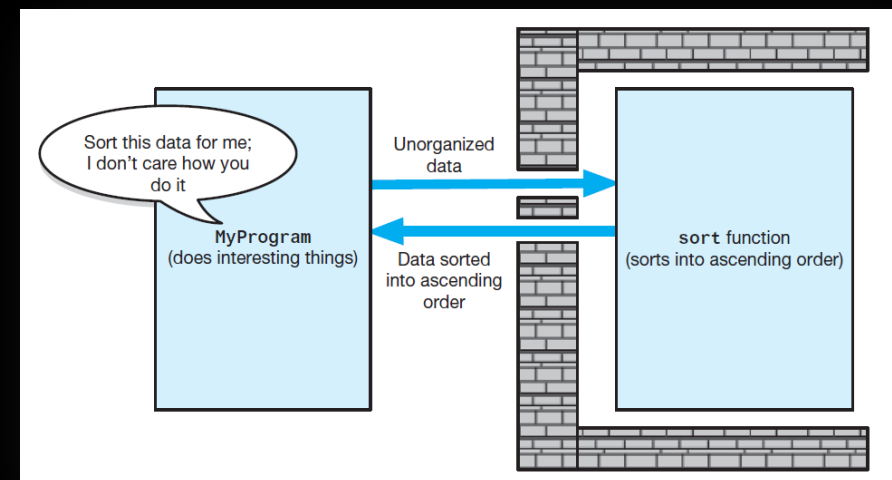
## Encapsulation

*Objects combine data and operations*



## Information Hiding

*Objects hide inner details*



## Inheritance

*Objects inherit properties from other objects*

## Polymorphism

*Objects determine appropriate operations at execution*

Coming soon

# Object-Oriented Solution

Use classes of objects

Combine **attributes** and **actions**

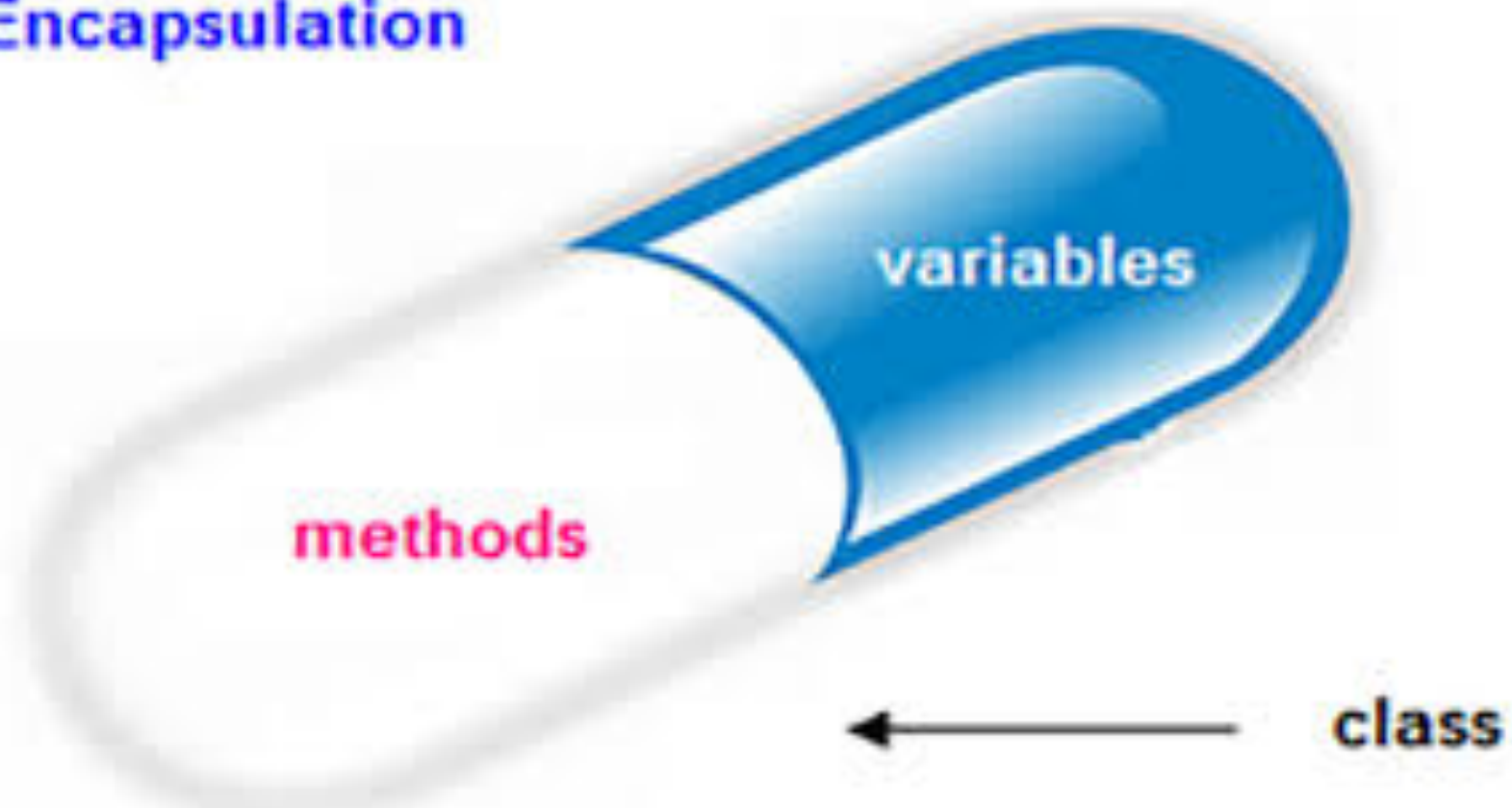
**data members** + **member functions**

Create a good set of **modules**

**Self contained unit of code**

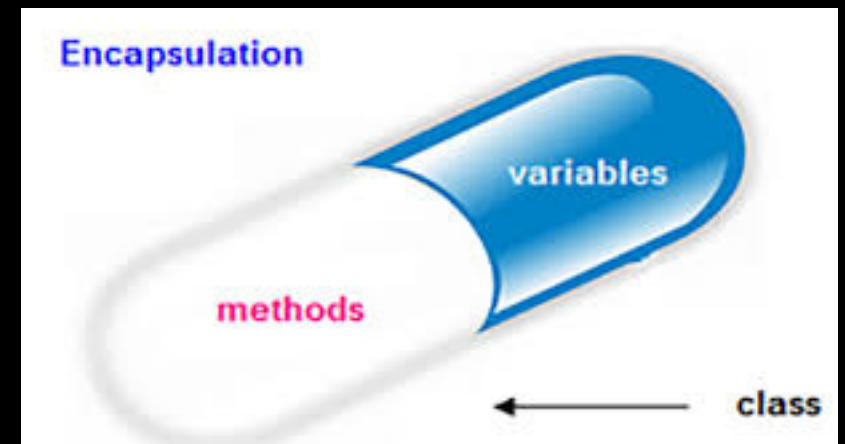
# Encapsulation

## Encapsulation





# Class



```
class SomeClass
{
    access_specifier    // can be private, public or protected

        data_members        // variables used in class

        member_functions    // methods to access data members

}; // end SomeClass
```

# Class

Language mechanism for

Encoding **abstraction**

Enforce **encapsulation**

Separate **interface** from **implementation**

You have already been  
working with classes.  
Which ones?

A **user-defined data type** that bundles together data  
and operations on the data

# Information Hiding

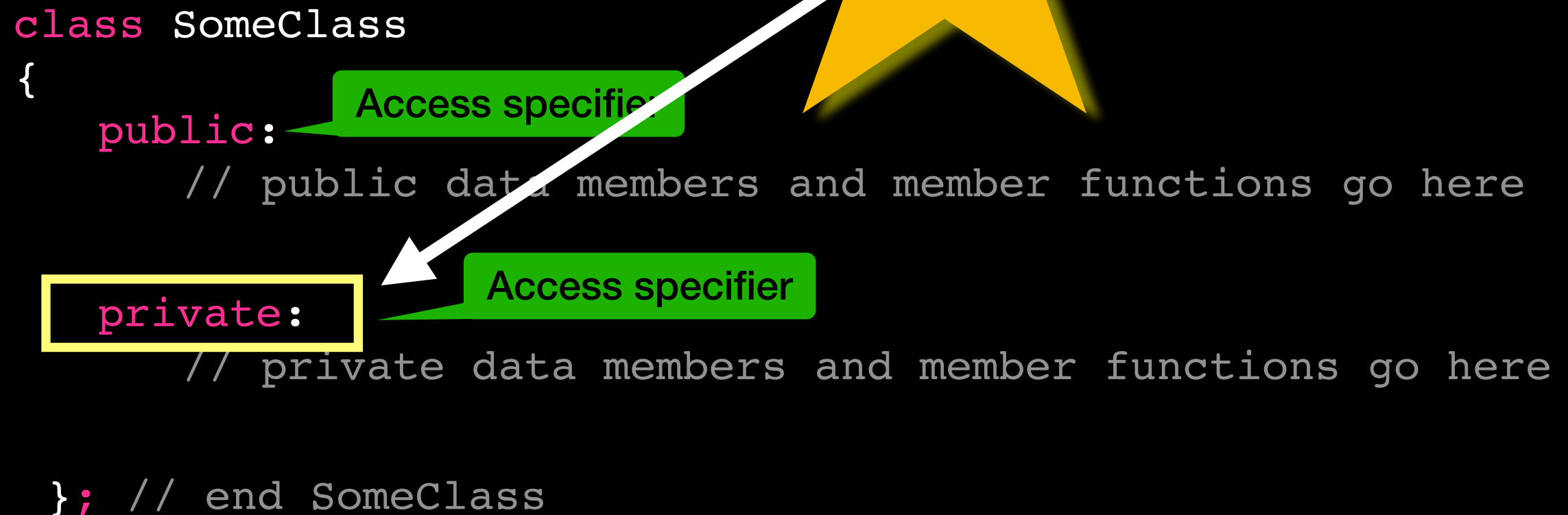
# Class

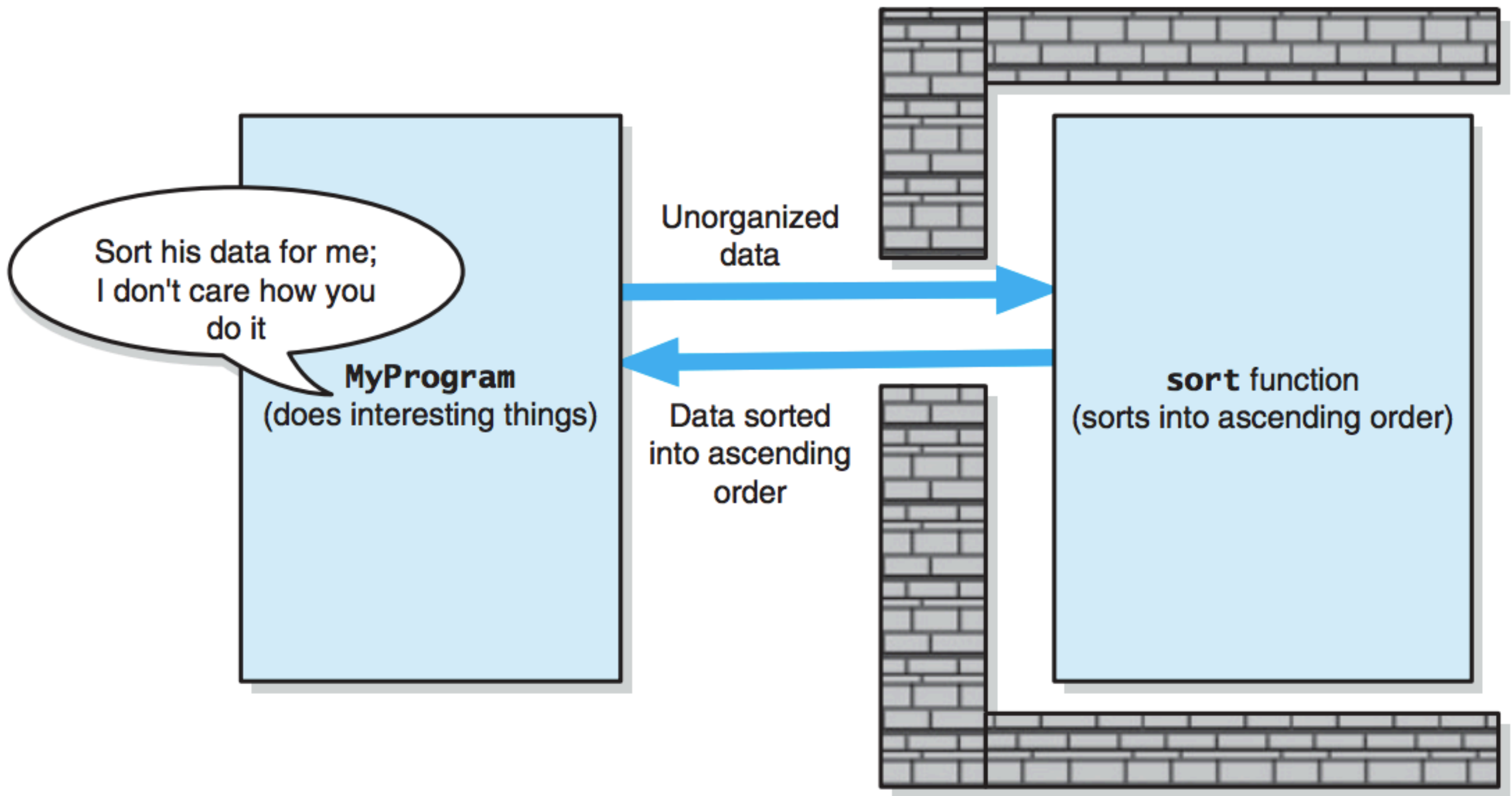
Information  
Hiding

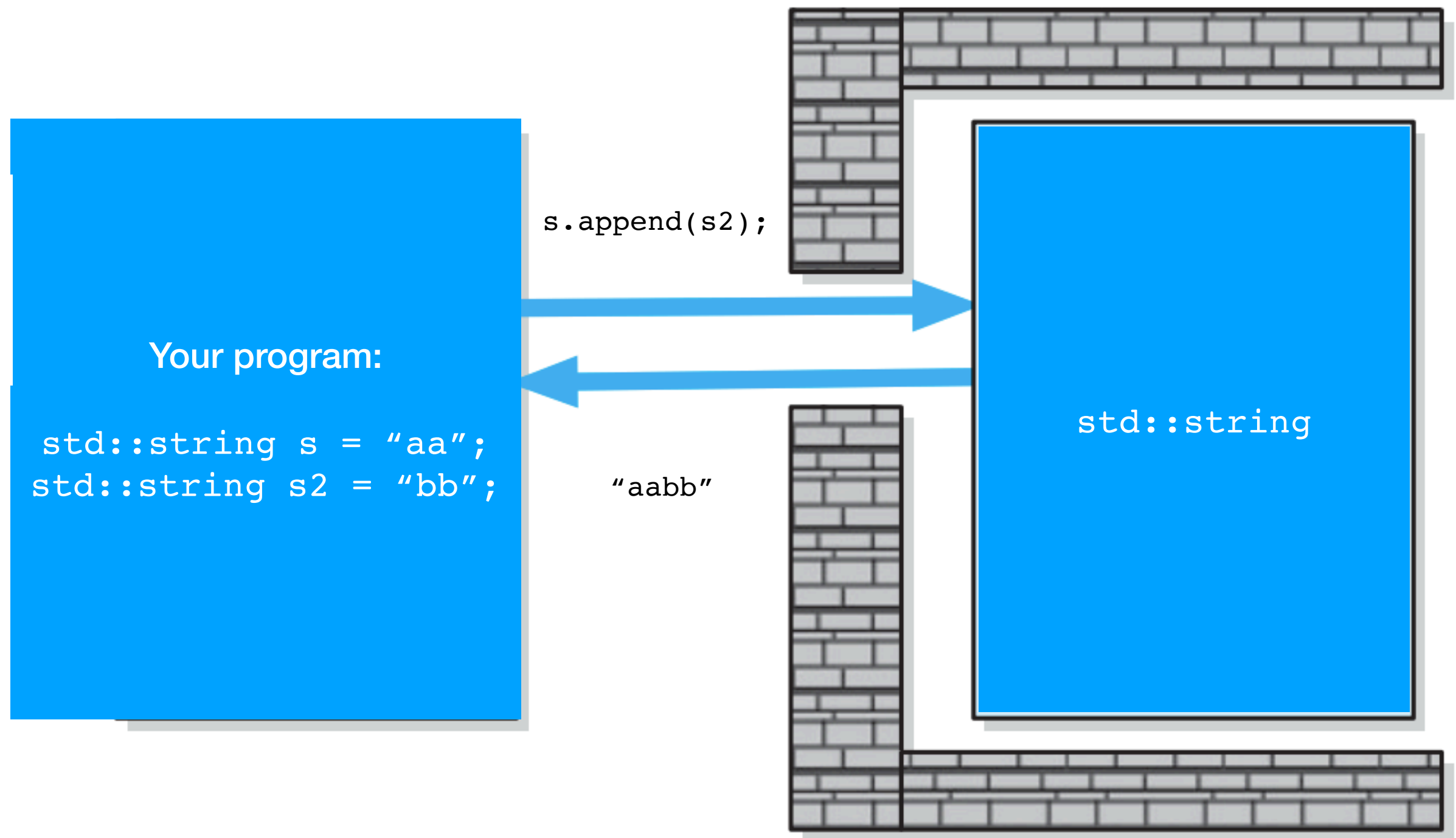
```
class SomeClass
{
    public:           Access specifier
        // public data members and member functions go here

    private:         Access specifier
        // private data members and member functions go here

}; // end SomeClass
```







# Interface

**SomeClass.hpp**  
(same as SomeClass.h)

```
#ifndef SOME_CLASS_HPP_
#define SOME_CLASS_HPP_

#include <somelibrary>
#include "AnotherClass.hpp"

class SomeClass
{
public:
    SomeClass(); //Constructor
    int methodOne();
    bool methodTwo();
    bool methodThree(int
                        someParameter);

private:
    int data_member_one_;
    bool data_member_two_;

};    //end SomeClass

#endif
```

# Implementation

**SomeClass.cpp**

```
#include "SomeClass.hpp"

SomeClass::SomeClass()
{
    //implementation here
}

int SomeClass::methodOne()
{
    //implementation here
}

bool SomeClass::methodTwo()
{
    //implementation here
}

bool SomeClass::methodThree(int
someParameter)
{
    //implementation here
}
```

# Interface

**SomeClass.hpp**  
(same as SomeClass.h)

```
#ifndef SOME_CLASS_HPP_
#define SOME_CLASS_HPP_

#include <somelibrary>
#include "AnotherClass.hpp"

class SomeClass
{
public:
    SomeClass(); //Constructor
    int methodOne();
    bool methodTwo();
    bool methodThree(int
                      someParameter);

private:
    int data_member_one_;
    bool data_member_two_;

}; //end SomeClass

#endif
```

## Include Guards:

Tells linker "include only if it has not been included already by some other module"

# Implementation

**SomeClass.cpp**

```
#include "SomeClass.hpp"

SomeClass::SomeClass()
{
    //implementation here
}

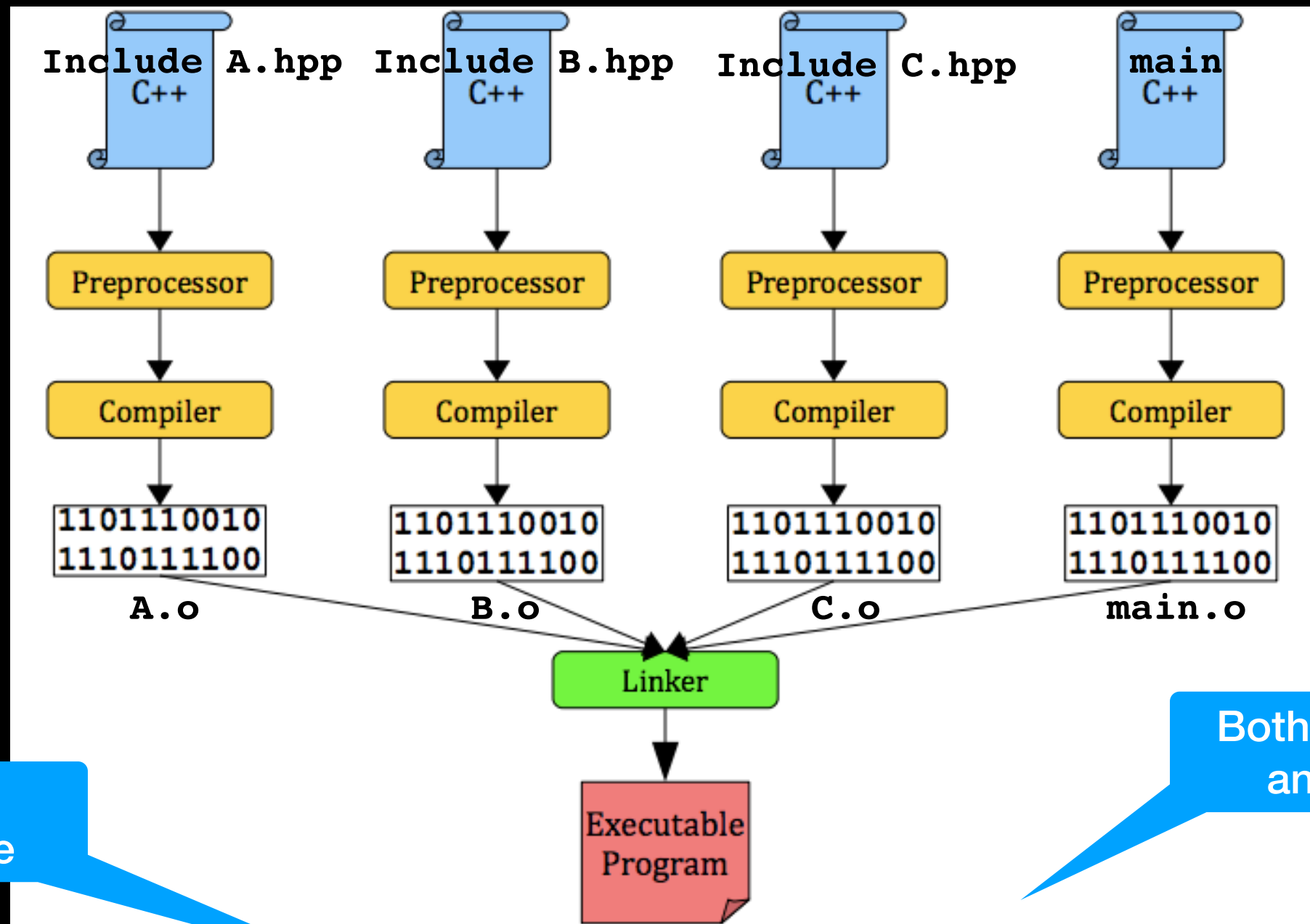
int SomeClass::methodOne()
{
    //implementation here
}

bool SomeClass::methodTwo()
{
    //implementation here
}

bool SomeClass::methodThree(int
someParameter)
{
    //implementation here
}
```



# Separate Compilation



```
g++ -o my_program A.cpp B.cpp C.cpp main.cpp
```

# Compile and Link separately with g++

```
g++ -c A.cpp B.cpp C.cpp main.cpp
```

will generate

```
A.o B.o C.o main.o
```

Then

```
g++ -o my_program A.o B.o C.o main.o
```

Will link the object files into a single executable named `my_program`

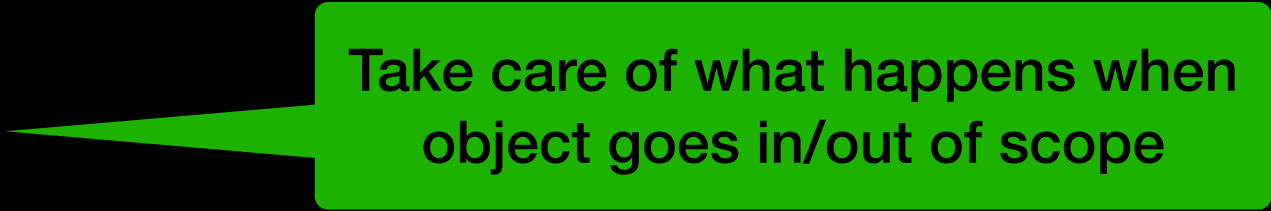
# Class Recap

**Access specifiers:** determines what data or methods are **public**, **private** or **protected** (more on protected later)

**Data members:** the attributes/data

**Member functions:** the operations/actions available on the data

- **Mutator functions:** modify data members
- **Accessor functions:** retrieve the value of data members  
Use `const` to enforce/indicate it will not modify the object  
e.g. `string getName() const;`
- **Constructor(s)**
- **Destructor**



Take care of what happens when object goes in/out of scope

# Class / Object

A class is a **user-defined data type** that bundles together data and operations on the data

**Class:** type (like `int`)

**Object:** instantiation of the class (like `x` - as in `int x`)

Just like variables, objects have a **scope**

- they are born (instantiated/**constructed**)



- they are killed (deallocated/**destroyed**)



# Object instantiation and usage

```
#include "SomeClass.h"
```

```
int main()  
{
```

Constructor is  
called here

```
SomeClass new_object; //instantiation of SomeClass calls constructor
```

```
int my_int_variable = new_object.methodOne();  
bool my_bool_variable = new_object.methodTwo();
```

```
return 0;
```

```
} //end main
```

object (dot) method  
calls the member function for this object

## DECLARATION / INTERFACE:

```
class SomeClass
{
```

```
public:
```

```
    SomeClass();
```

//default constructor

```
    SomeClass( parameter_list );
```

//parameterized constructor

```
    // public data members and member functions go here
```

```
private:
```

```
    // private members go here
```

```
}; // end SomeClass
```

# Constructors



**Default Constructor** automatically supplied by compiler if no constructors are provided. Primitive types are initialized to 0

If only Parameterized Constructor is provided, compiler **WILL NOT** supply a Default Constructor and class **MUST** be initialized with parameters

**Executed when object is declared.**  
Initializes member variables and does whatever else may be required at instantiation

## DECLARATION / INTERFACE:

```
class SomeClass
{
```

```
    public:
```

```
        SomeClass();
```

//default constructor

```
        SomeClass( parameter_list );
```

//parameterized constructor

```
        // public data members and member functions go here
```

```
    private:
```

```
        // private members go here
```

```
}; // end SomeClass
```

# Constructors



## IMPLEMENTATION:

```
SomeClass::SomeClass()
```

```
{
```

```
// end default constructor
```

```
SomeClass::SomeClass(type parameter_1, type parameter_2):
```

```
member_var1(parameter_1), member_var2(parameter_2)
```

```
{
```

```
//end parameterized constructor
```

OR:

```
SomeClass::SomeClass():
```

```
member_var1(initial value),
```

```
member_var2(initial value)
```

```
{
```

```
// end default constructor
```

**Member Initializer List**



## DECLARATION / INTERFACE:

```
class SomeClass
{
    public:
        SomeClass() = default;           //default constructor
        SomeClass(parameter_list)      //parameterized constructor
        // public data members and member functions go here

    private:
        // private members go here
}; // end SomeClass
```

C++ 11

Tells compiler to provide  
default constructor!

## IMPLEMENTATION:

```
SomeClass::SomeClass(type parameter_1, type parameter_2):
    member_var1(parameter_1), member_var2(parameter_2)
{
} //end parameterized constructor
```



# Destructor



Default Destructors automatically supplied by compiler if not provided.

**Must** provide Destructor to free-up memory when SomeClass performs dynamic memory allocation

```
class SomeClass
{
    public:
        SomeClass();
        SomeClass( parameter_list ); //parameterized constructor
        // public data members and member functions go here
        ~SomeClass(); // destructor

    private:
        // private data members and member functions go here
}; // end SomeClass
```

Executed when object goes out of scope or explicitly deleted to release memory

# Lecture Activity

**Write the interface for a printer class:**

```
class Printer
{
    access_specifier    // can be private, public or protected

    data_members        // variables used in class

    member_functions    // methods to access data members

}; // end Printer
```

# Interface as Operation Contract

Documents use and limitations of a class and its methods

Function Prototype and Comments **MUST** specify:

- Data flow
  - Input => parameters
  - Output => return
- Pre and Post Conditions

# Operation Contract

In Header file:

```
/** sorts an array into ascending order
// @pre 1 <= number_of_elements <= MAX_ARRAY_SIZE
// @post an_array[0] <= an_array[1] <= ...
//      <= an_array[number_of_elements-1];
//      number_of_elements is unchanged
// @param an_array of values to be sorted
// @param number_of_elements contained in an_array
// @return true if an_array is sorted, false otherwise
*/
bool sort(const int& an_array[], int number_of_elements);
```

**Function prototype**

# Back to some principles of Software Engineering

# Unusual Conditions

Values out of bound, null pointer, inexistent file...

How to address them (strive for fail-safe programming):

State it as precondition

Return value that signals a problem

Typically a boolean to indicate success or failure

Throw an exception (later in semester)

# Solution guidelines

Many possible designs/solutions

Often no clear best solution

“Better” solution principles:

*High cohesion*

*Loose Coupling*

# Cohesion

Performs one well-defined task

Well named => self documenting

e.g. `sort()`

**SORT ONLY!!!**

E.g. If you want to output,  
do that in another function

Easy to reuse

Easy to maintain

Robust (less likely to be affected by change)



# Coupling

Measure of *dependence (interactions)* among modules

i.e. share data structures or call each other's methods

Minimize but cannot eliminate  
Objects must collaborate!!!



# Reduce Coupling

Methods should only call other methods:

- defined within **same class**
- of **argument** objects
- of objects **created within** the method
- of objects that are **data members** of the class

# Control Interaction

Pass-by-value

```
bool my_method(int some_int);
```

Pass-by-reference if need to modify object

```
bool my_method(ObjectType& some_object);
```

Pass-by-constant-reference if function doesn't modify object

```
bool my_method(const ObjectType& some_object);
```



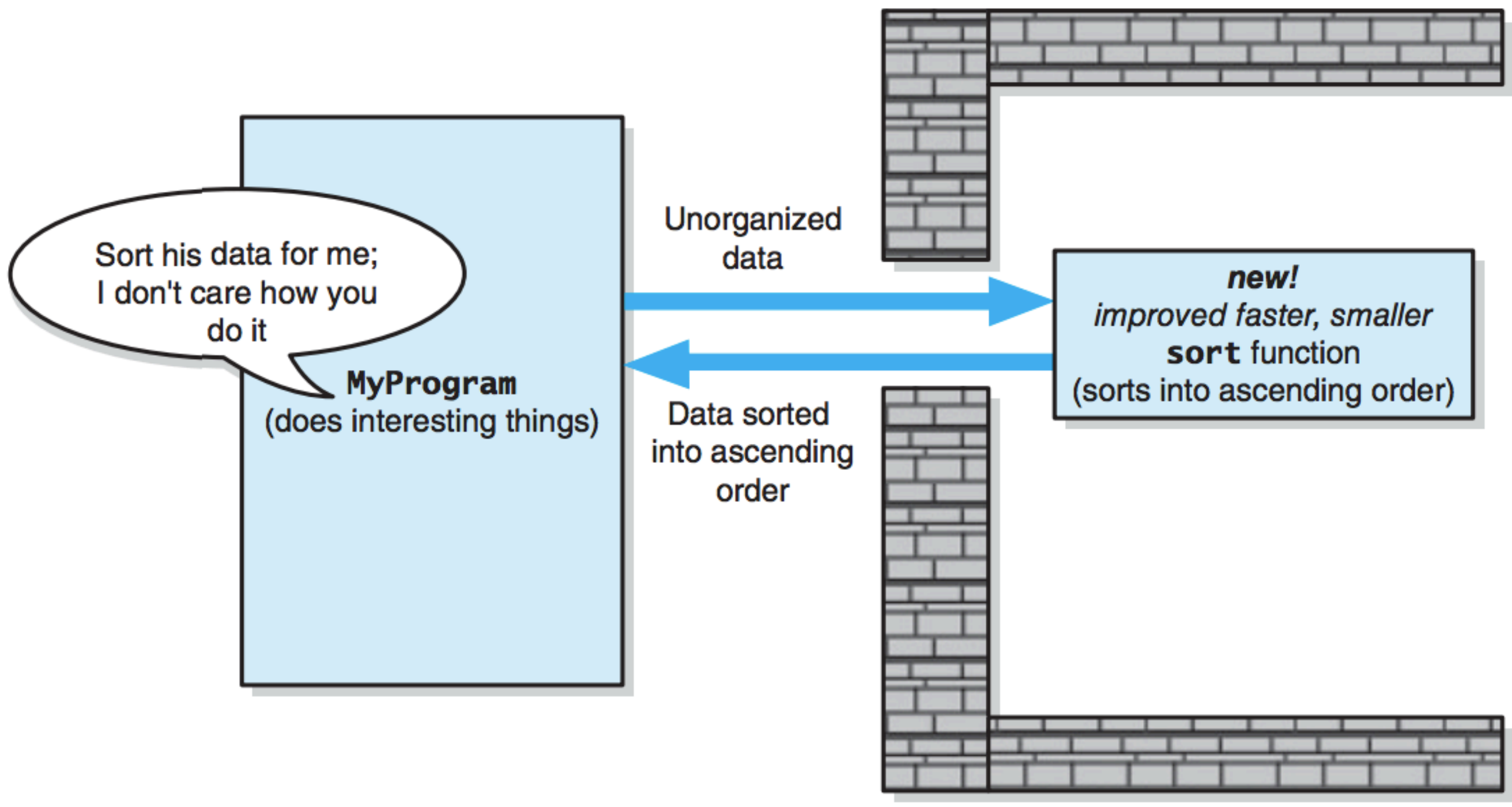
# Modifiability

No global variables EVER!!!

## Named Constants

```
const int NUMBER_OF_MAJORS = 160;  
int scores [NUMBER_OF_MAJORS];  
for(index = 0 through NUMBER_OF_MAJORS - 1)  
    Process
```

# Modifiability



# Readability

**BAD!!!**

Write **self-commenting** code

Important to strike balance btw readable code and comments

- don't write the obvious in comments

```
x += m * v1; //multiply m by v1 and add result to x
```

Use descriptive names for variables and methods

```
/**@return: the average of values in scores*/
double getAverage(double* scores, int size)
{
    double total = 0;

    for (int i = 0; i < size; i++)
    {
        total += scores[i];
    }

    return ( total / (double)size );
}
```

# Naming Conventions

<https://google.github.io/styleguide/cppguide.html>

<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#R1-comments>

```
string my_variable;
```

or

```
string myVariable;
```

**Classes ALWAYS**

**start with capital**

```
MyClass
```

In this course I will strive for:

```
class MyClass
```

```
MyClass class_instance;
```

```
string my_variable;
```

```
string my_member_variable_;
```

```
void myMethod();
```

```
int MY_CONSTANT;
```



**Be consistent!!!**

# Project 1

Animal class (baseline):

- implement a trivial class
- compile multi-file program with g++ (Animal.cpp, main.cpp)
- submit to Gradescope

Projects 1-4 build on each other, do not fall behind!