

Abstraction and OOP

Tiziana Ligorio

tligorio@hunter.cuny.edu

Today's Plan



Announcements

Recap

Abstraction

OOP

Announcements and Syllabus Check

Email change: tligorio@hunter.cuny.edu

Office hours **1001A** (unless you see a note on the door and/or announcement on course webpage)

Check the course webpage for UTA tutoring schedule

Review grading policy

YOU MUST CHECK YOUR HUNTER EMAIL

Project 1

Getting started on the right foot

Review / Baseline

No surprises

Be proactive

<https://tligorio.github.io/Project1.html>

Recap

Minimize complexity

Simplify complex program to manageable level

Break down into smaller problems

Isolate functionalities

Abstraction

Abstraction Example



Abstraction Example



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

Abstraction Example

Easy to use

Come in all shapes and sizes

Can have different mechanisms

Complex inner-mechanism

What makes a stapler?

What makes a stapler?

Staplers fasten paper together

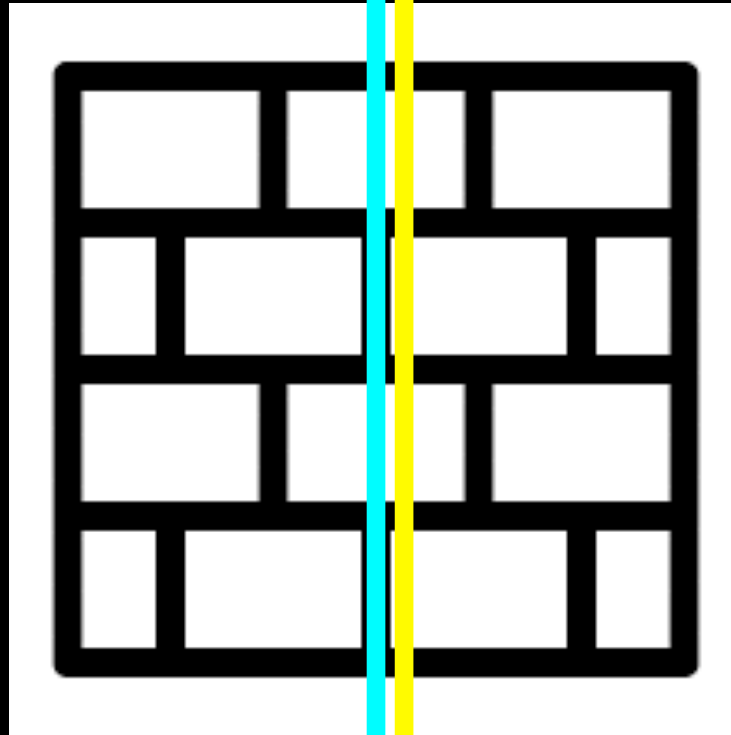
Separate functionality from implementation
(i.e. what can be done from how it's actually done)

Wall of Abstraction

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

Painstaking work to design and implement stapler to work smoothly and correctly

Design and implementation



Press handle
Or
Feed paper near sensor

Usage

Abstractions are imprecise

A stapler fastens paper together

Wall of abstraction between *implementer* and *client*

How does client know how to use it?

Provide an *interface*

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

In-Class Task

Attributes:

Actions:

Interface for Stapler

Attributes:

Number of staples left

Size of staples being used

Max number of sheets of paper can be stapled together

Actions:

Staple paper together

Put paper into stapler

Add more staples



How this is done
is irrelevant to
the client

Information Hiding

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 and directly modify **objects**

Reasons for Information Hiding

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

Reduces flexibility and modifiability by locking implementation in place

Increases complexity of interactions between modules

So back to software

Reduce Complexity

Abstraction and **Information Hiding** are a means for entirely containing complexity

Immense amount of implementation detail is abstracted away into a **small set of commands** executed by means of an **interface**

Information hiding prevents parts of a program from inadvertently (or deliberately) modifying implementations in unexpected ways



Object Oriented Programming (OOP)

Object Oriented Analysis and Design (OOAD)

Problem solving

Problem statement => Solution

Solution: computer program specified as a system of interacting classes of objects

Object-oriented *analysis* specifies

What to do (requirements), not how to do it

Object-oriented *design* specifies

Software objects and their collaboration

Object-Oriented Solution

Create a good set of modules

Self contained unit of code

Use classes of objects

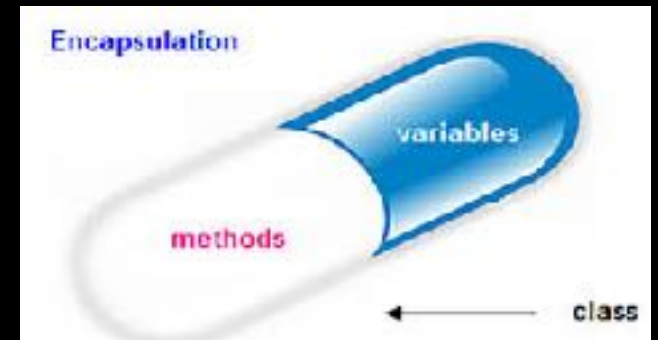
Combine attributes and behaviors

data members + member functions

Principles of Object Oriented Programming (OOP)

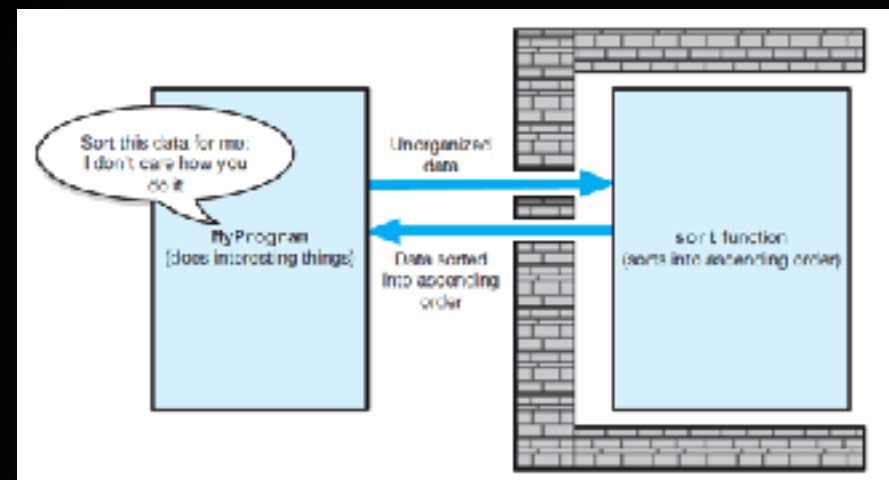
Encapsulation

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

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!!!



Minimize
complexity

“Ok, this is all great. But how do I do it?”

Class

Language's mechanism for

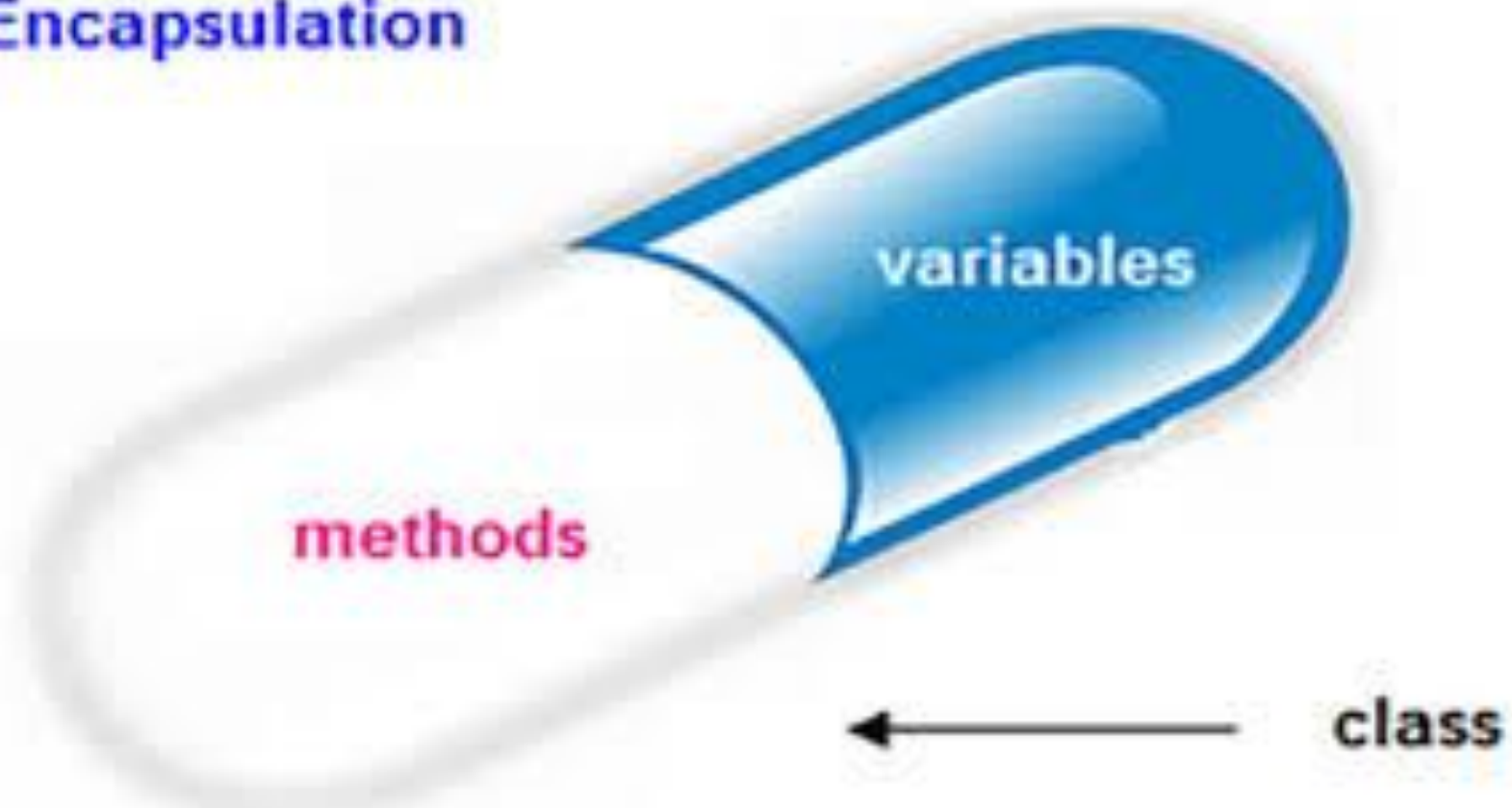
Encoding **abstraction**

Enforce **encapsulation**

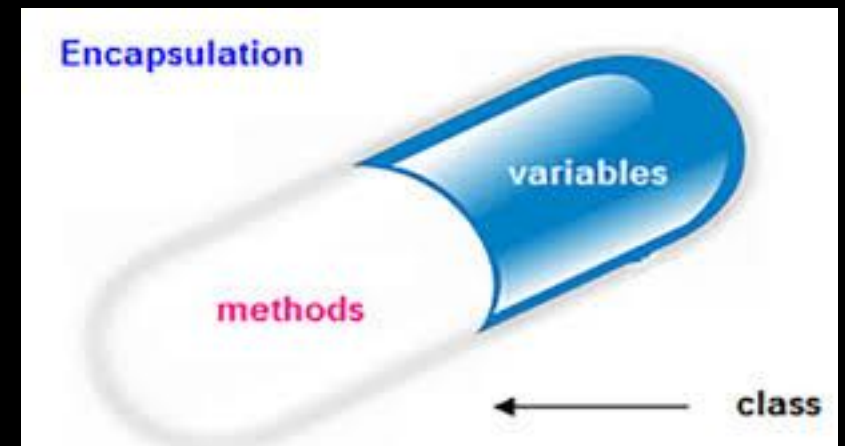
Pairing **interface** with **implementation**

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
```

Information Hiding


Class

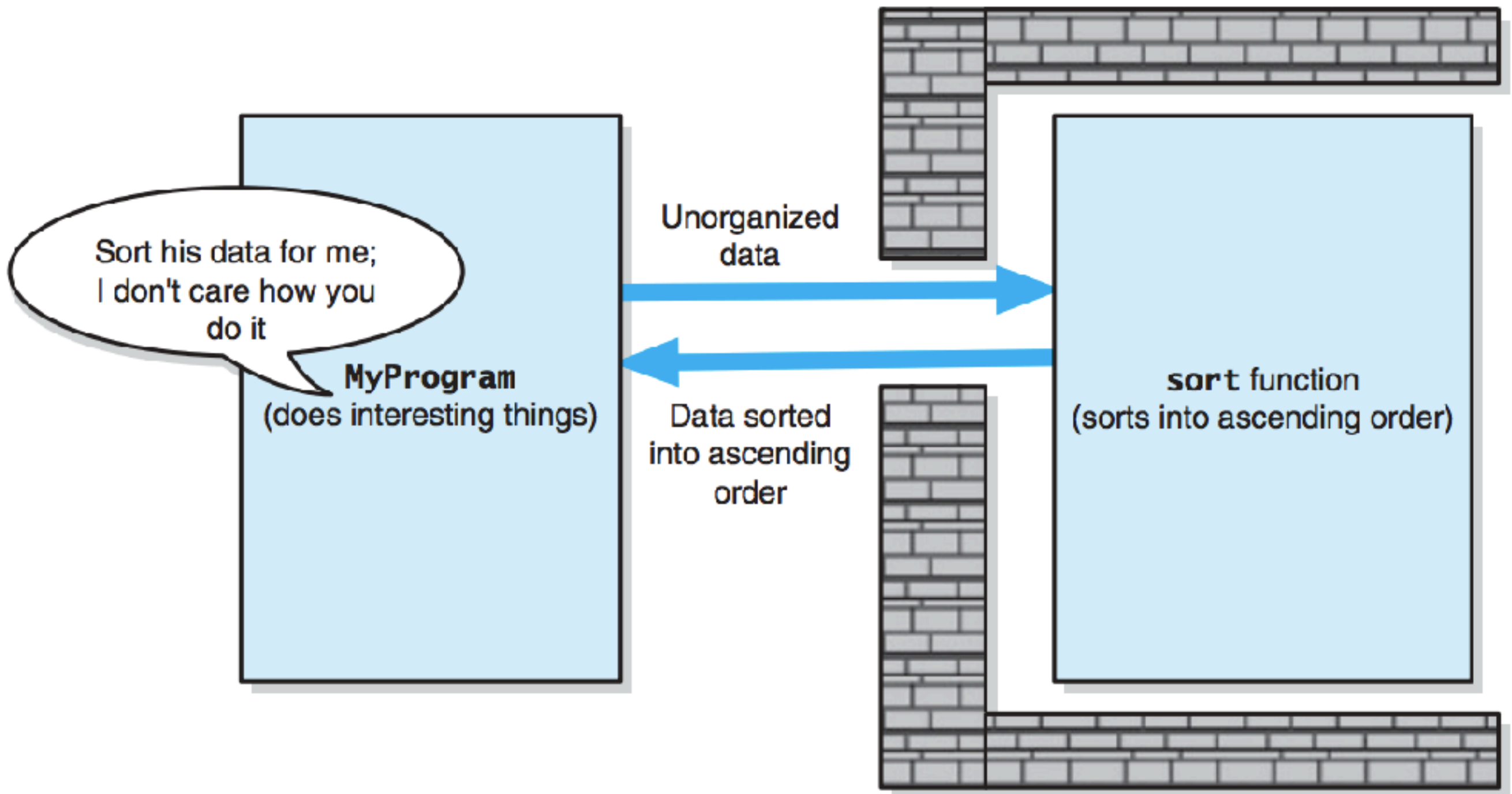
Information
Hiding

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

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

};           // end SomeClass
```

A white arrow points from the 'private:' access specifier in the code to the 'Information Hiding' text inside a yellow star.



In-Class Taks

Write a stapler class:

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

    data_members        // variables used in class

    member_functions    // methods to access data members

};                      // end Stapler
```

Interface

Header file!!!!

Same as .h

SomeClass.h (or SomeClass.hpp)

Public member *prototype* (function declaration)

```
bool sort(const int& an_array[], int number_of_elements);  
return type + descriptive name + ( parameter list )
```

Operation Contract

```
// these are this method's assumptions and what it does  
// I will not tell you how it does it!!!
```

Operation Contract

Documents use and limitations of a method

Specifies

Data flow (Input and Output)

Pre and Post Conditions



Comments above functions in header file

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
// return: true if an_array is sorted, false otherwise
bool sort(const int& an_array[], int number_of_elements);
```



Function prototype

Style

Comments should be **helpful** not **redundant**

Interface comments should say **what** the function does not **how** it does it

```
//add new_customer to genius_bar_ and update their wait time  
//return: true if customer added successfully, false otherwise  
bool addWaitingCustomer(Customer& new_customer);
```

As opposed to

```
//add newCustomer to genius_bar_ and update their wait time  
//return: true if number_of_customers_ <= MAX_NUMBER_OF_CUSTOMERS, false otherwise  
bool add_waiting_customer(Customer& new_customer);
```

Unusual Conditions

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

How to address them:

State it as precondition

Return value that signals a problem

Typically a boolean to indicate success or failure

Throw an exception (later in semester)

