ADTs & Templates

Tiziana Ligorio
tligorio@hunter.cuny.edu

Assessment Quiz

Today's Plan

Recap

ADTs

Templates

Intro to Inheritance

Announcements and Syllabus Check

Announcements on course webpage - Questions?

Linux accounts processed this morning

Lecture slides AFTER class

Communication: **csci235.help@gmail.com**
(we may just skip the blackboard forum at this point)

Pointers and References (start ahead if you are not familiar with them!!!)

Syllabus: still on track, but could be running behind after today

Opportunities

Tech Talent Pipeline
https://huntercuny2x.github.io/
Application deadline soon (9/15 - 9/30):
https://cunyhunter.co1.qualtrics.com/jfe/form/SV_bNlA08EDYSsn03z

CUNY Tech Prep (for next year)
https://cunytechprep.nyc/

Recap

OPP

- Abstraction
- Encapsulation
- Information Hiding

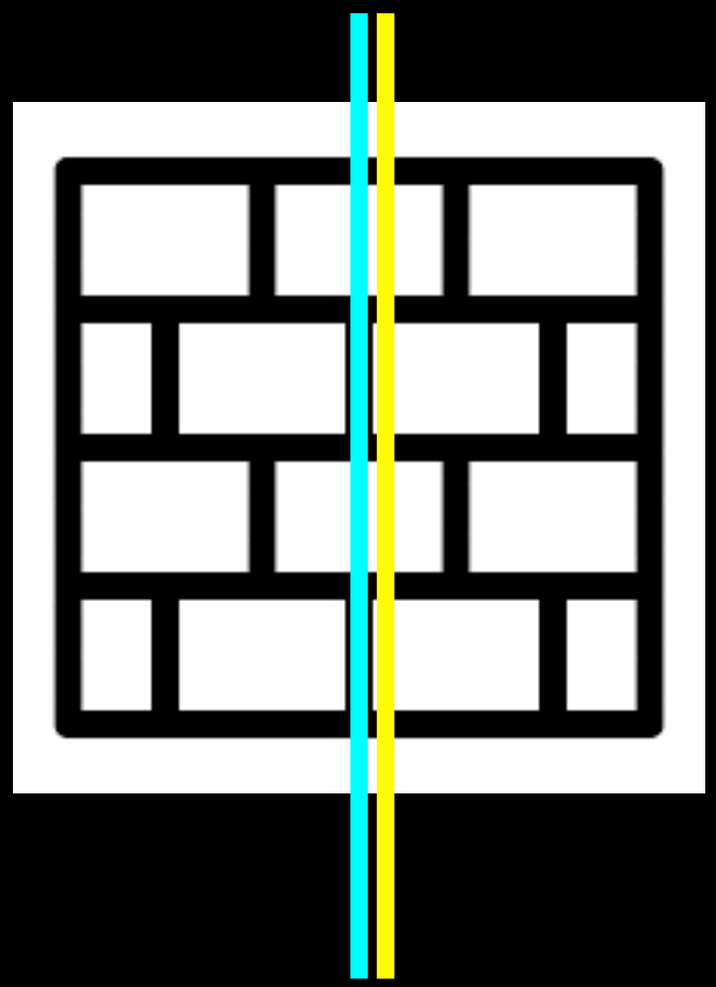
Classes

- Public Interface
- Private Implementation
- Constructors / Desctructors
- Overloading operators

Wall of Abstraction

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

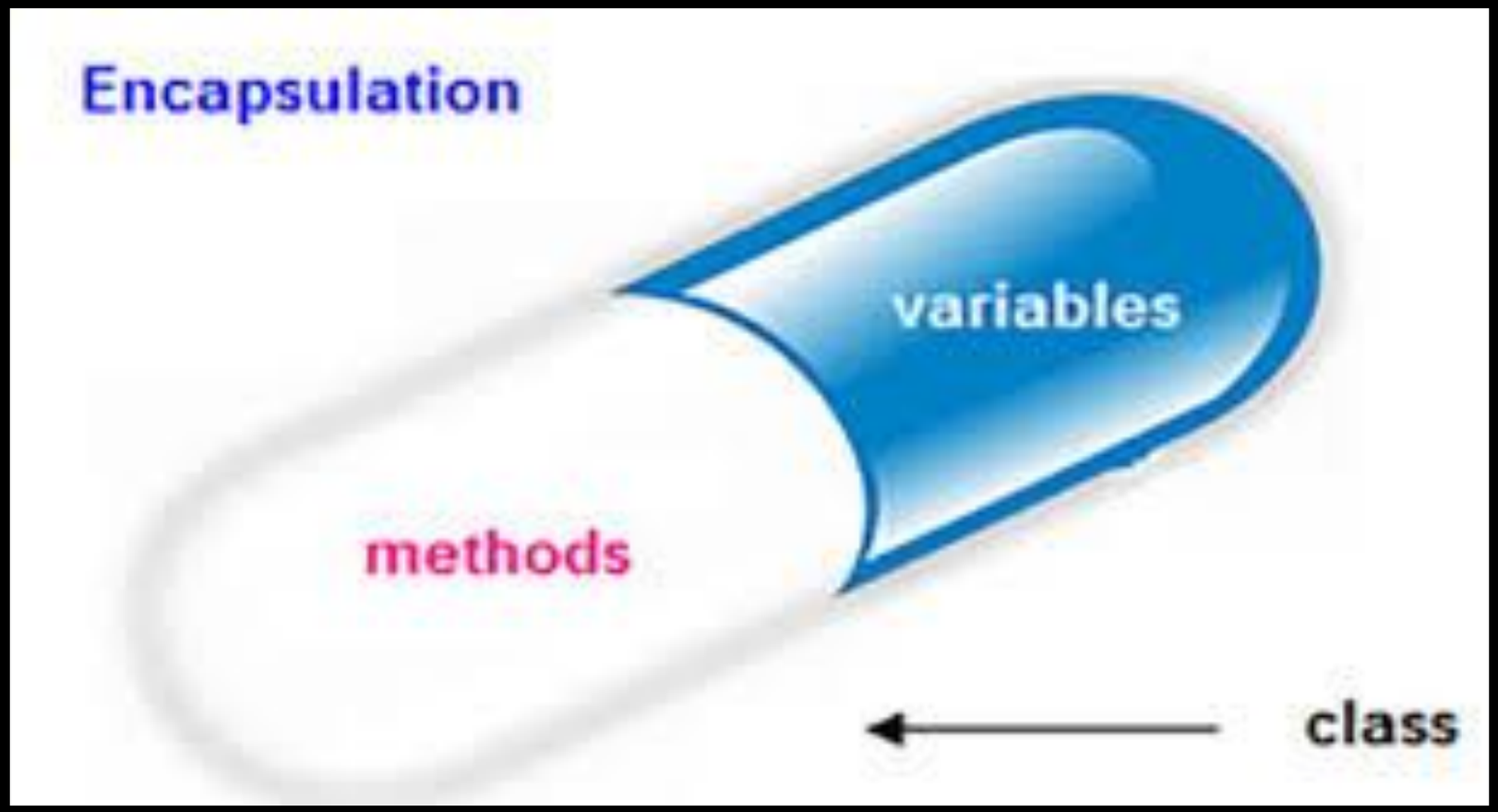
Painstaking work to design and implement stapler to work smoothly and correctly



Press handle
Or
Feed paper near sensor

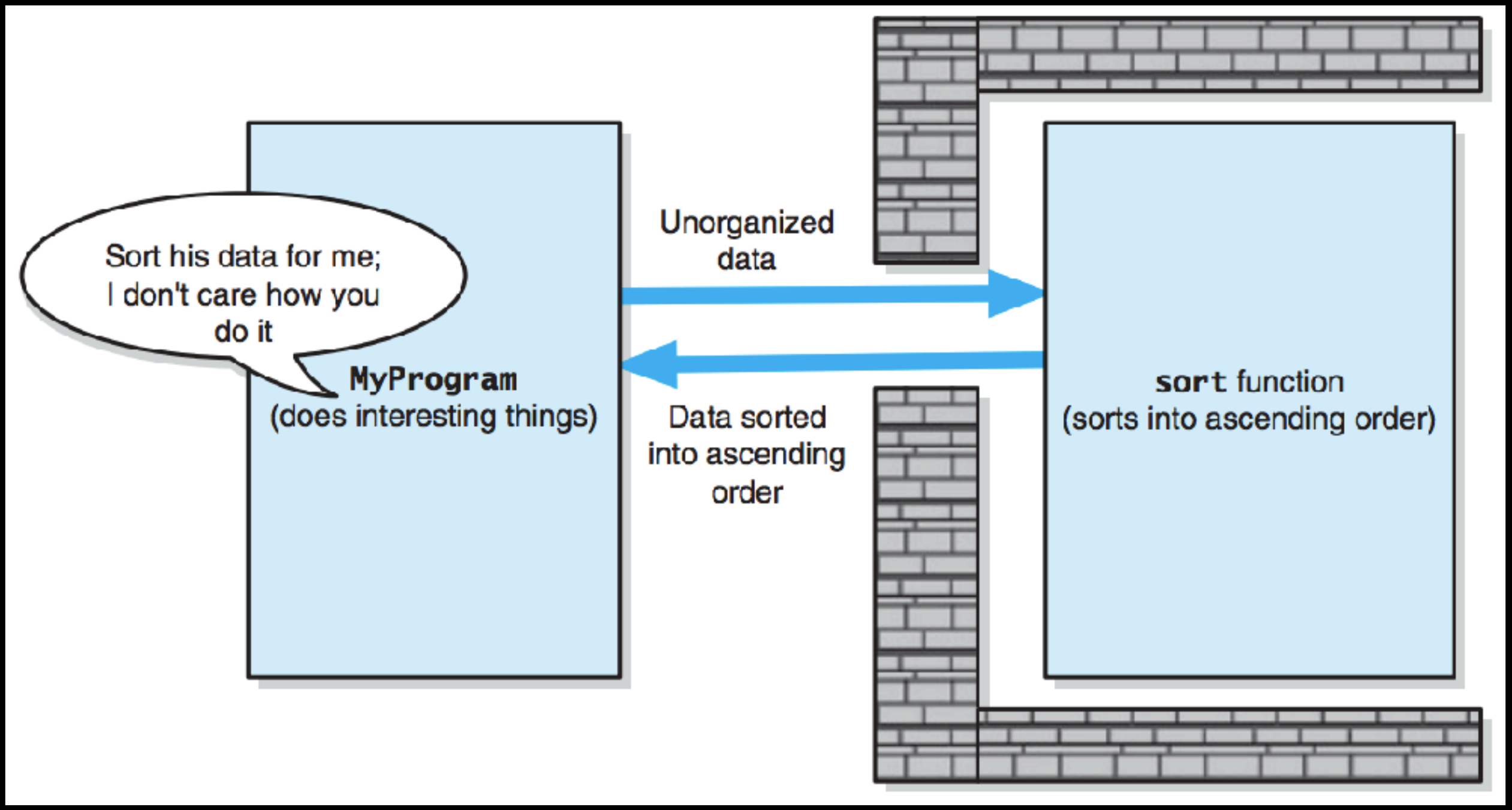
Design and implementation

Usage


Encapsulation
variables
methods
class

Class


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

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

};                          // end SomeClass
```


Sort his data for me; I don't care how you do it
MyProgram
(does interesting things)
Unorganized data
Data sorted into ascending order
sort function
(sorts into ascending order)

Interface

Header file!!!!

`SomeClass.h (or SomeClass.hpp)`

Same as .h

Public member ***prototype*** (function declaration)

```
bool sort(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(int& an_array[], int number_of_elements);
```

Function prototype

Unusual Conditions

Values out of bound, null pointer, inexistent file…

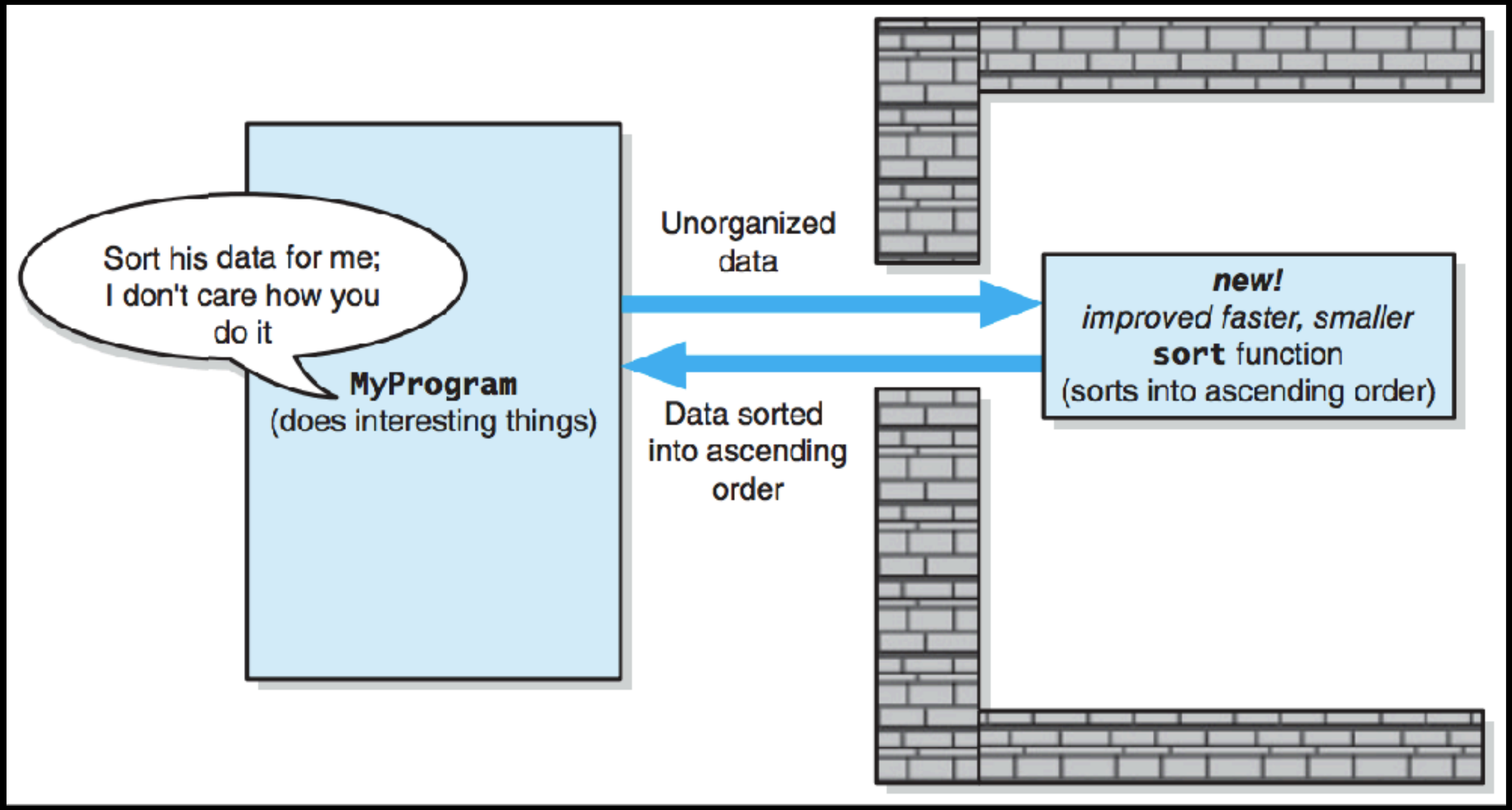
How to address them:

State it as precondition

Return value that signals a problem

Typically a boolean to indicate success or failure

Throw an exception (later in semester)


Sort his data for me;
I don't care how you
do it
MyProgram
(does interesting things)
Unorganized
data
Data sorted
into ascending
order
new!
improved faster, smaller
sort function
(sorts into ascending order)

Constructors

DECLARATION:

```
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
```

Default Constructor automatically supplied by compiler if not provided.

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

Constructors

DECLARATION:

```
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
```

Default Constructor automatically supplied by compiler if not provided.

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

IMPLEMENTATION:

```
SomeClass::SomeClass()
{
}// end default constructor
```

OR:

```
SomeClass::SomeClass():
member_var1_(initial value),
member_var2_(initial value)
{
}// end default constructor
```

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

Member Initializer List

Destructors

Default Destructors automatically supplied by compiler if not provided.

Must provide Destructor to free-up memory when SomeClass does 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
```

Overloading Functions

Same name, different parameter list (different function prototype)

```
int someFunction()
{

//implementation here

} // end someFunction


int someFunction(string
some_parameter )
{
    //implementation here

}   // end someFunction
```

```
int main()
{

    int x = someFunction();

    int y = someFunction(my_string);

    //more code here

} // end main
```

Friend Functions

Functions that are not members of the class but CAN access private members of the class

Violates Information Hiding!!!

Yes, so don't do it unless appropriate and controlled

Friend Functions

DECLARATION:

```
class SomeClass
{
    public:
        // public data members and member functions go here
        friend returnType someFriendFunction( parameter list);
    private:
        // private data members and member functions go here

};                          // end SomeClass
```

IMPLEMENTATION (`SomeClass.cpp`):

Not a member function

```
returnType someFriendFunction( parameter list)
{
    // implementation here
}
```

Operator Overloading

Desirable operator (=, +, -, == …) behavior may not be well defined on objects

```
class SomeClass
{
    public:
        // public data members and member functions go here
        friend bool operator== (const SomeClass& object1,
                                const SomeClass& object2);


    private:
        // private members go here

  };    // end SomeClass
```

Operator Overloading

IMPLEMENTATION (`SomeClass.cpp`):

Not a member function

```
bool operator==(const SomeClass& object1,
                const SomeClass& object2)
{
    return ( (object1.memberA_ == object2.memberA_) &&
             (object2.memberB_ == object2.memberB_) && … );
}
```

Default Arguments

Order Matters!
Parameters without default arguments must go first.

```
void point(int x = 3, int y = 4);

point(1,2); // calls point(1,2)
point(1);   // calls point(1,4)
point();    // calls point(3,4)
```

Similarly:

```
Customer(string name, string device = "unknown", int wait_time = 0);

Customer(“Lina”); // calls Customer(“Lina”,”unknown”, 0)
Customer(“Gina”, “iPhone”);   // calls Customer(“Gina”,”iPhone”, 0)
Customer(“Nina”, “iPad”, 5);   // calls Customer(“Nina”,”iPad”, 5)
```

Abstract Data Type

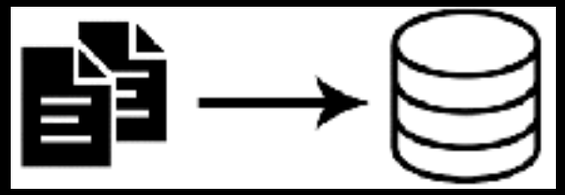
Data and Abstraction

Operations on data are central to most solutions

Think abstractly about data and its management

Typically need to

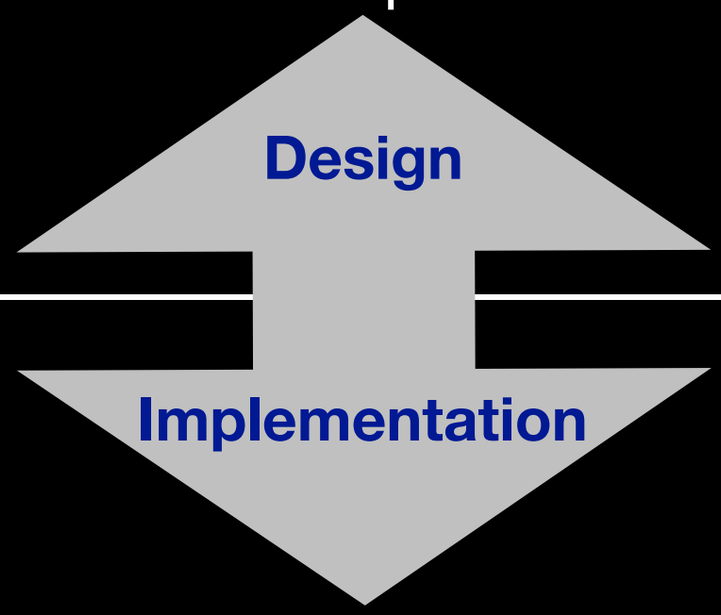
- Add data
- Remove data
- Retrieve
- Reorganize data
- Ask questions about data
- Modify data



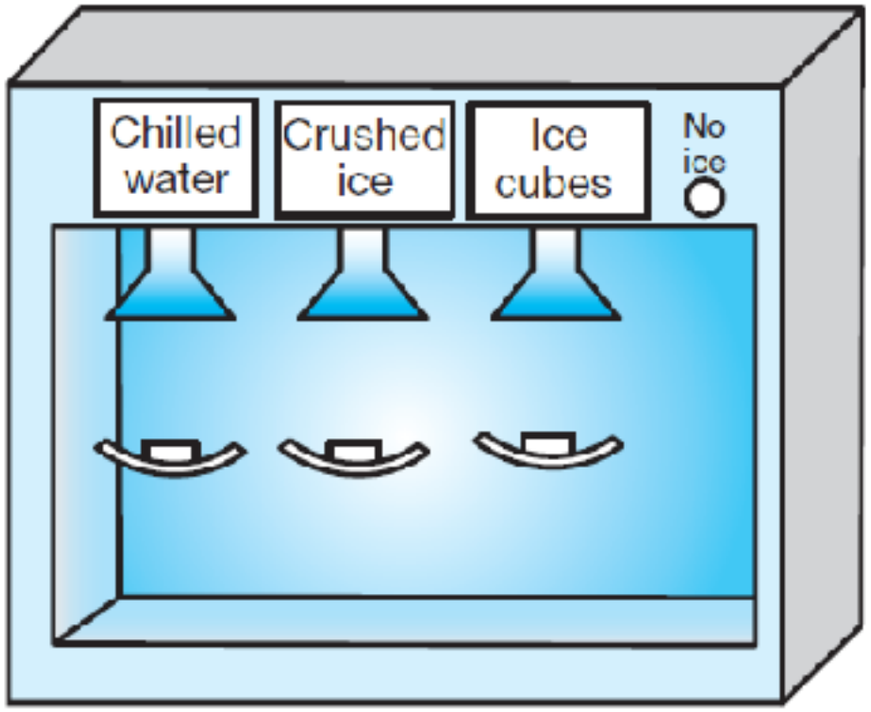
Abstract Data Type

A collection of data and a set of operations on the data

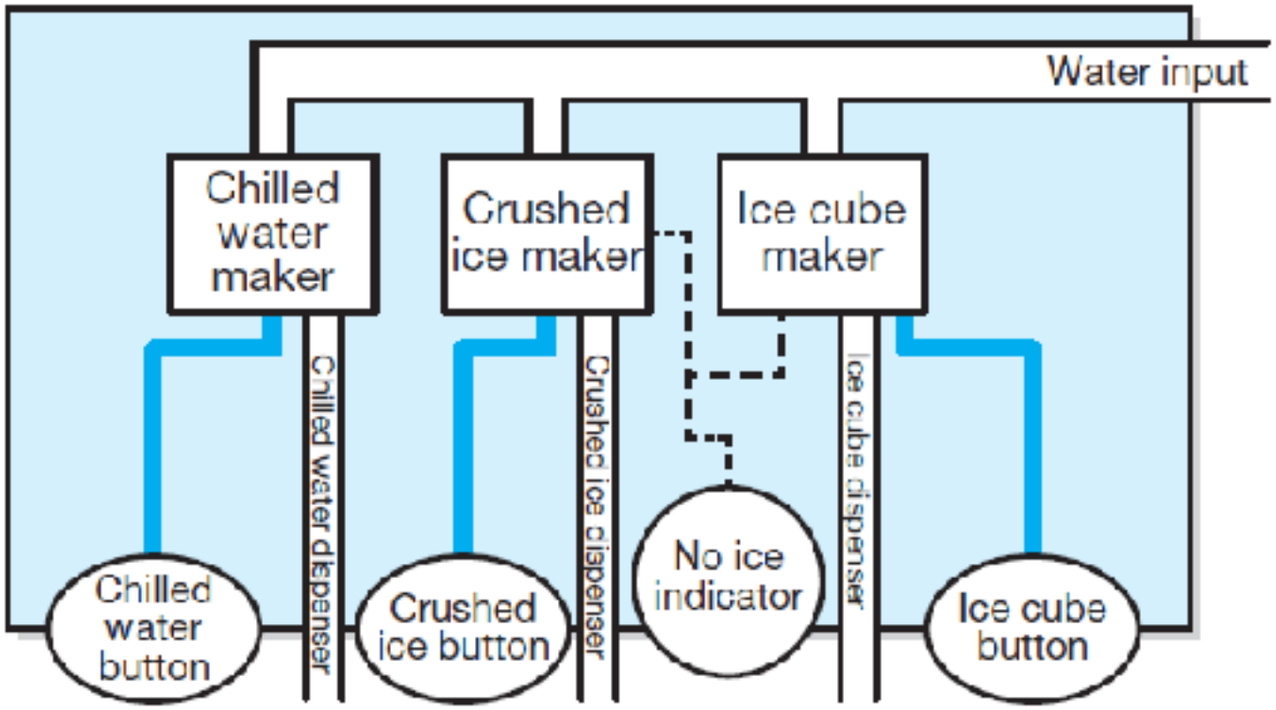
Carefully specify and ADT's operations before you implement them



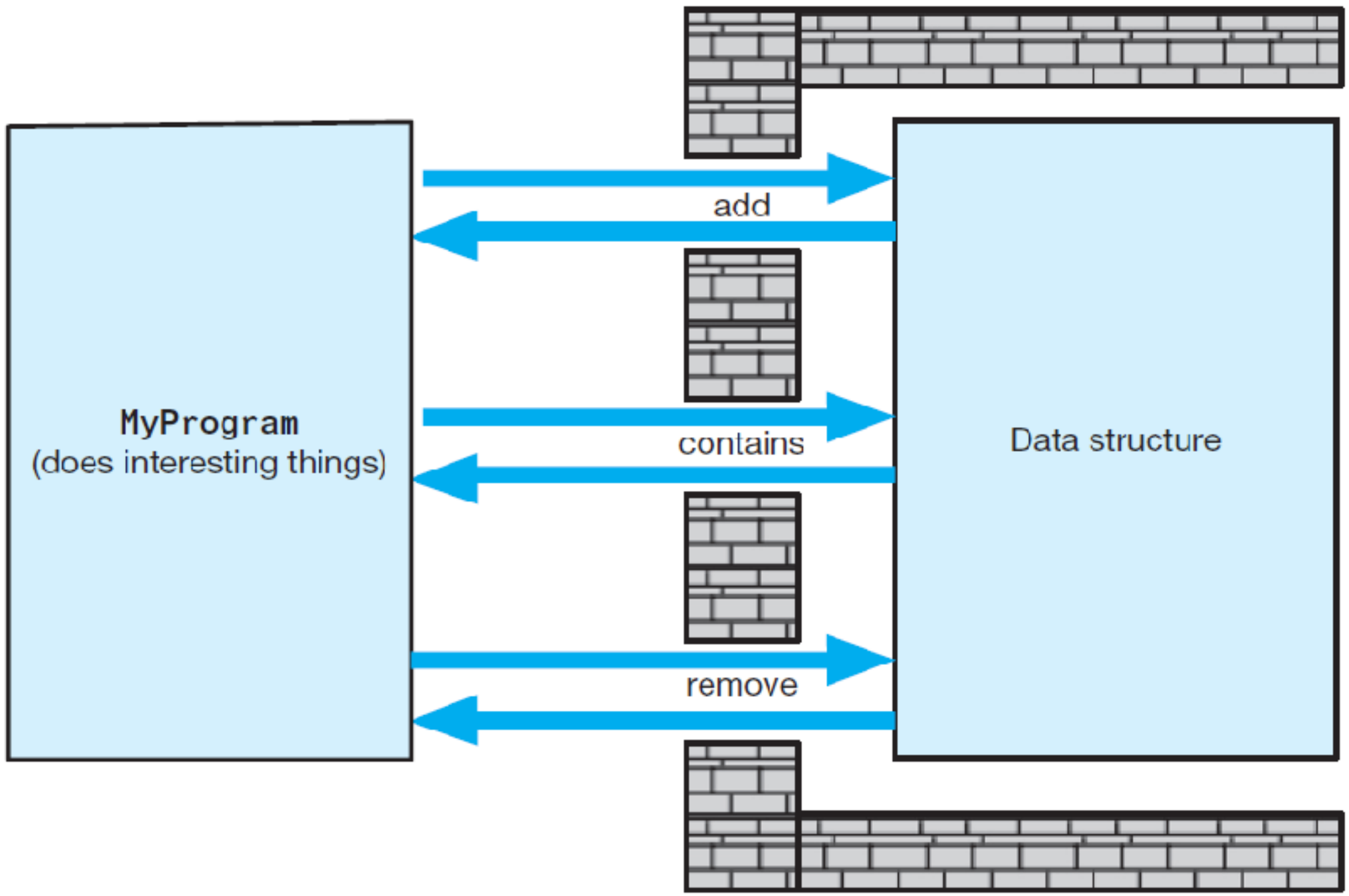
In C++ member variables and member functions implement the Data Structure



User's exterior view

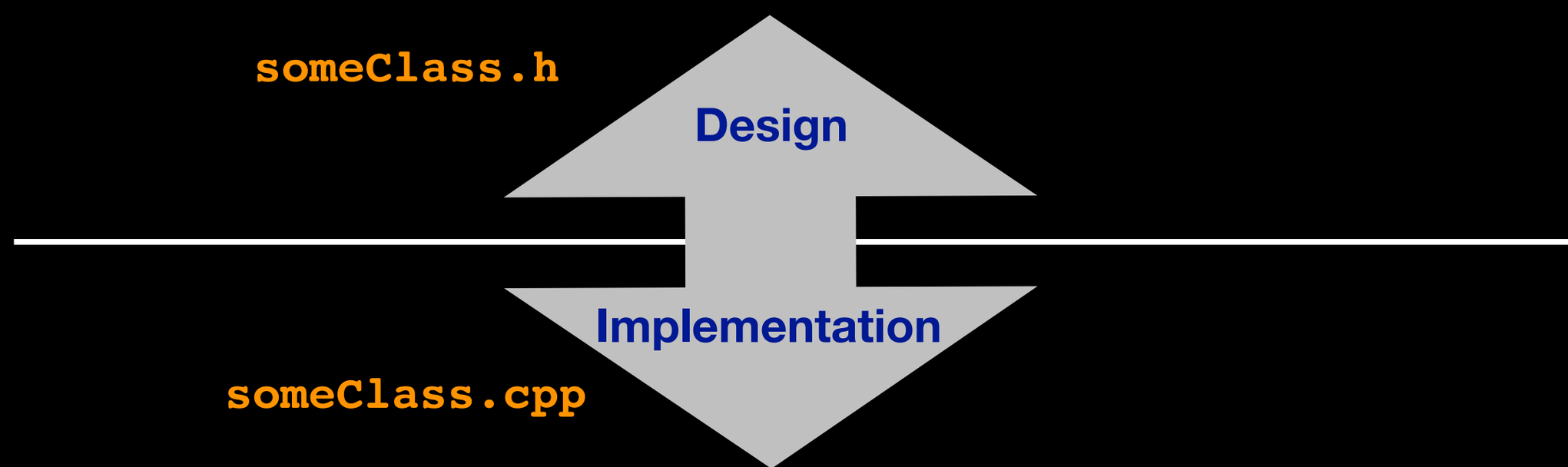


Technician's interior view


MyProgram
(does interesting things)
add
contains
remove
Data structure

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
```



Designing an ADT

What data does the problem require?

- Names
- IDs
- Numerical data

What operations are necessary on that data?

- Initialize
- Display
- Calculations
- Add
- Remove
- Change

Design the Bag ADT

Contains things

Container or Collection of Objects

Objects are of same type

No particular order

Can contain duplicates

In-class Task

Bag Operations:

1.
2.
3.
4.
5.
6.
…

Identify Behaviors

Bag Operations:

1. Get the number of items currently in the bag
2. See whether the bag is empty
3. Add an object to the bag
4. Remove an occurrence of a specific object form the bag, if possible
5. Remove all objects from the bag
6. Count the number of times a certain object is found in the bag
7. Test whether the bag contains a particular object
8. Look at all the objects that are in the bag

Specify Data and Operations

Pseudocode

```
//Task: reports the current number of objects in Bag
//Input: none
//Output: the number of objects currently in Bag
getCurrentSize()

//Task: checks whether Bag is empty
//Input: none
//Output: true or false according to whether Bag is empty
isEmpty()

//Task: adds a given object to the Bag
//Input: new_entry is an object
//Output: true or false according to whether addition succeeds
add(new_entry)

//Task: removes an object from the Bag
//Input: an_entry is an object
//Output: true or false according to whether removal succeeds
remove(an_entry)
```

Specify Data and Operations

```
//Task: removes all objects from the Bag
//Input: none
//Output: none
clear()

//Task: counts the number of times an object occurs in Bag
//Input: an_entry is an object
//Output: the int number of times an_entry occurs in Bag
getFrequencyOf(an_entry)

//Task: checks whether Bag contains a particular object
//Input: an_entry is an object
//Output: true of false according to whether an_entry is in Bag
contains(an_entry)

//Task: gets all objects in Bag
//Input: none
//Output: a vector containing all objects currently in Bag
toVector()
```

Vector

A container similar to a one-dimensional array

Different implementation and operations

STL (C++ standart template library)

```
#include <vector>
…
std::vector<type> vector_name;
```

e.g.

```
std::vector<string> student_names;
```

In this course cannot use STL for projects unless specified so by instructions

What's next?

Finalize the interface for your ADT => write the actual code

But we have a problem

What's next?

Finalize the interface for your ADT => write the actual code

But we have a problem

We said Bag contains objects of same type

What type?

To specify member function prototype we need to know

```
//Task: adds a given object to the Bag
//Input: new_entry is an object
//Output: true or false according to whether addition succeeds
bool add(type??? new_entry);
```

Templates

Motivation

We don’t want to write a new Bag ADT for each type of object we might want to store

Want to parameterize over some arbitrary type

Useful when implementing an ADT without locking the actual type

An example are STL containers

e.g. `vector<type>`

Declaration

```
#ifndef BAG_H_
#define BAG_H_
template<class ItemType> // this is a template definition
class Bag
{

    //class declaration here

}
#include "Bag.cpp"
#endif //BAG_H_
```

Explained next

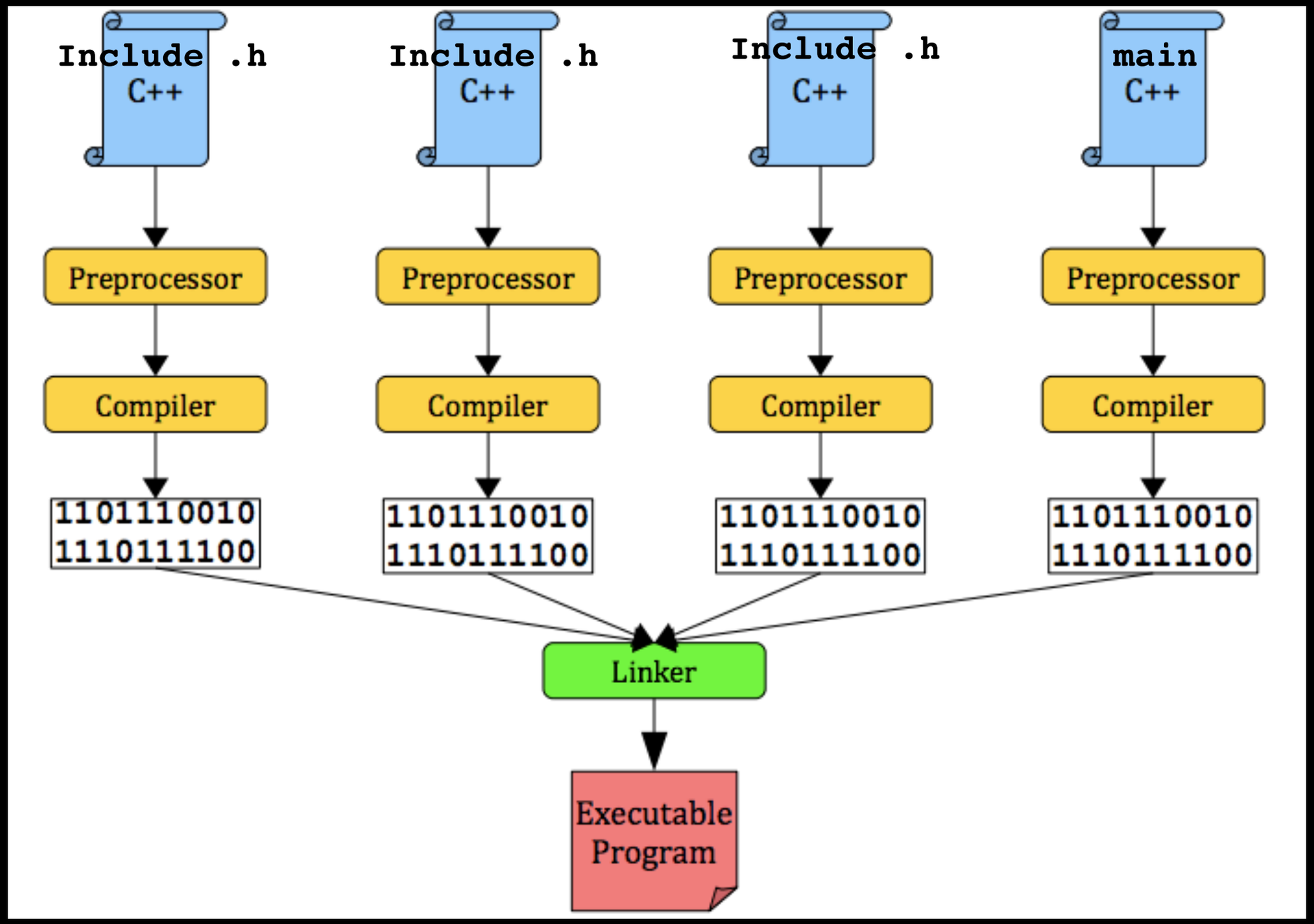
Implementation

```
#include "Bag.h"

template<class ItemType>
bool Bag<ItemType>::add(const ItemType& newEntry){
    //implementation here
}

    //more member function implementation here
```

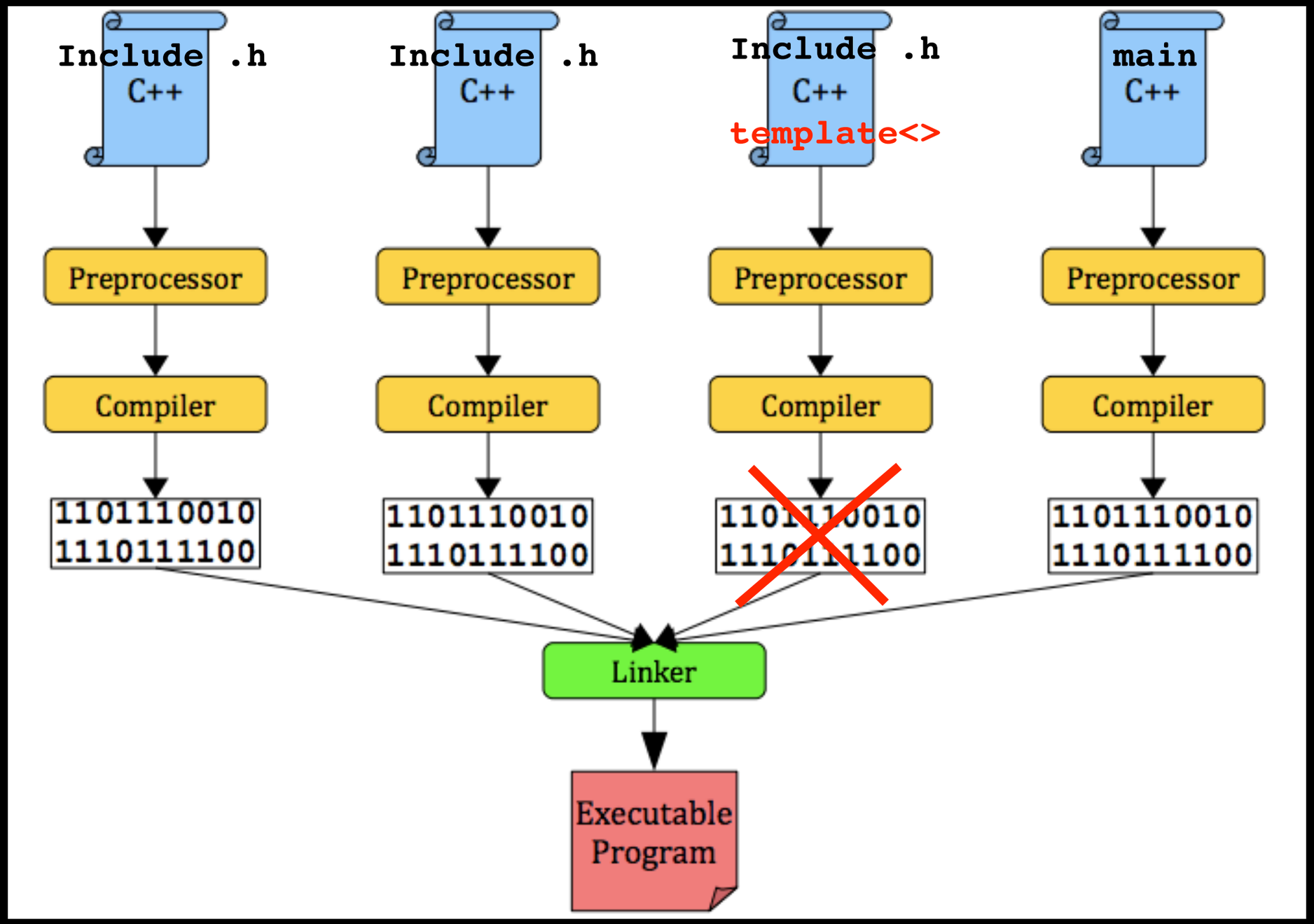
Instantiation

```
#include "Bag.h"

int main()
{

    Bag<string> stringBag;
    Bag<int> intBag;
    Bag<someObject> someObjectBag;

    //stuff here

    return 0;

}; // end main
```

Linking with Templates



Linking with Templates



Linking with Templates

Always `#include` the `.cpp` file in the `.h` file

```
#ifndef MYCLASS_H_
#define MYCLASS_H_

//stuff here

#include "MyClass.cpp"
#endif //MYCLASS_H_
```

Do not add `MyClass.cpp` to project and do not include it in the command to compile

```
g++ -o my_program main.cpp
```

Not `g++ -o my_program MyClass.cpp main.cpp`

Programming Practice

Write a simple templated dummy class `MyTemplate`

Give it some functionality, e.g:

- a parameterized constructor that initializes some private data member `my_data_` of type `ItemType`
- an accessor member function `getData()`

Write a `main()` function that initializes different `MyTemplate` objects with different types (e.g. `int`, `string`) and makes calls to their accessor member functions to observe their behavior. E.g:

```
MyTemplate<int> intObject;
cout << intObject.getData() << endl;
```

Make sure you understand and don't have problems with multi-file compilation using templates

Back to `Bag`

"Now I'm still in the design phase… I feel pretty good about my design… I have a general container, I have decided to templatize it so I can store any type in it… but I feel I could do something more with it"

–Stream of Thought

Bag

From General to Specific

Can envision needing another container with all Bag functionalities + extra features

Don’t want to re-write all the code for Bag+

In-class Task

Assume I have written all the code for Bag and I have been using Bag objects in many of my programs. It is a general container and it has work well so far. Now I need to write a new program, I need a container very similar to Bag, but more specific, it also needs to keep items sorted.

Assume you are all-powerful, and not limited to your current knowledge of C++ but can let the language do whatever you want (maybe you are Bjarne Stroustrup and you are writing C++)

What would you do?

Inheritance

From General to Specific

What if we could *inherit* functionality from one class to another?

We can!!!

Inherit `public` members of another class

Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void changeCartridge();
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```

Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void changeCartridge();
    void printDocument(const string& document);
private:
    // stuff here
};  //end Printer
```

```
class BatchPrinter
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
};  //end BatchPrinter
```

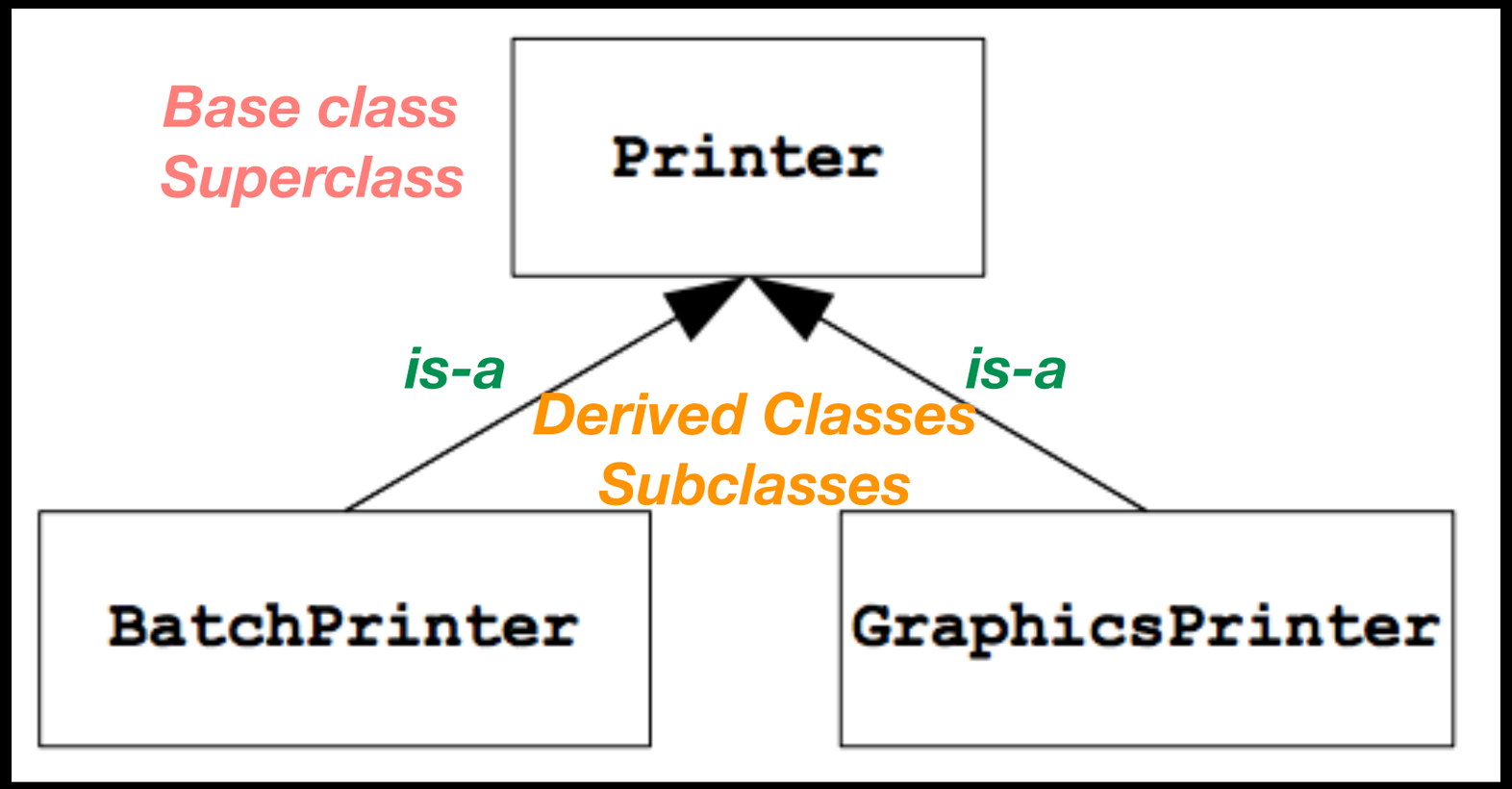
Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void changeCartridge();
    void printDocument(const string& document);
private:
    // stuff here
};  //end Printer
```

***Inherited members are** `public` **could be** `private` **or** `protected` – more on this later*

```
class BatchPrinter: public Printer// inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
};  //end BatchPrinter
```

Basic Inheritance




```
void initializePrinter(Printer& p) //some initialization function
BatchPrinter batch;
initializePrinter(batch); //legal because batch is-a printer
```

Think of argument types as specifying minimum requirements

Overloading vs Overriding

Overloading (independent of inheritance): Define new function with same name but different parameter list (different signature or prototype)

```
int someFunction(){ }
int someFunction(string some_string){ }
```

Overriding: Rewrite function with *same signature* in derived class

```
int BaseClass::someMethod(){ }
int DerivedClass::someMethod(){ }
```

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void changeCartridge();
    void printDocument(const string& document);
private:
    // stuff here
};  //end Printer
```

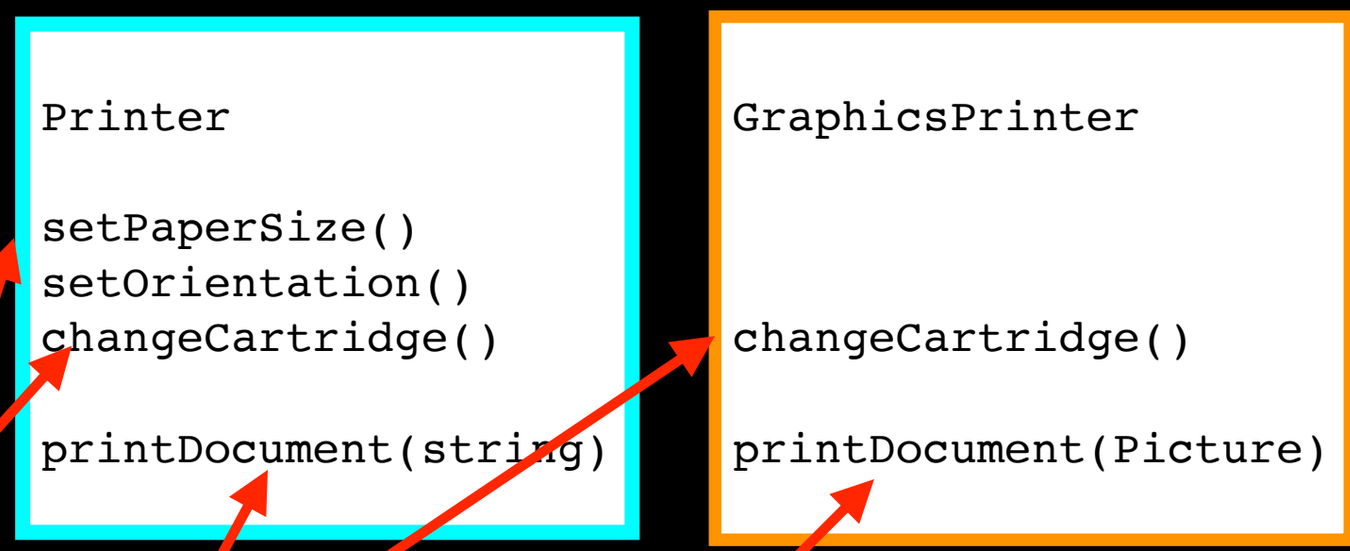
```
class GraphicsPrinter: public Printer // inherit from printer
{
public:
    //Constructor, destructor
    void changeCartridge();
    void printDocument(const Picture& picture);//for some Picture
                                               //object

private:
    //stuff here
};  //end GraphicsPrinter
```

Overrides `changeCartridge()`

Overloads `printDocument()`

main()

```
Printer basePrinter;
GraphicsPrinter graphicsPrinter;
Picture picture;
// initialize picture here
string document;
// initialize document here

basePrinter.changeCartridge(); //calls Printer function
graphicsPrinter.changeCartridge(); // Overriding!!!
graphicsPrinter.setPaperSize(); //inherited

graphicsPrinter.printDocument(string); //calls Printer inherited function
graphicsPrinter.printDocument(picture); // Overloading!!!
```



protected access specifier

```
class SomeClass
{
   public:
      // public members available to everyone

   protected:
      // protected members available to class members
      // and derived classes

   private:
      // private members available to class members ONLY

};                          // end SomeClass
```

Important Points about Inheritance

Derived class inherits all `public` and `protected` members of base class

Does not have access to base class `private` members

Does not inherit constructor and destructor

Does not inherit assignment operator

Does not inherit friend functions and friend classes

Constructors

A class needs user-defined constructor if must initialize data members

Base-class constructor always called before derived-class constructor

If base class has only parameterized constructor, derived class **must** supply constructor that calls base-class constructor explicitly

Constructors

```
class BaseClass()
{
public:
    //stuff here

private:
    //stuff here
 }; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    //stuff here
}; //end DerivedClass

DerivedClass::DerivedClass()
{
    //implementation here
}
```

```
main()
```

```
DerivedClass my_derived_class;
//BaseClass default constructor called
//then DerivedClass constructor called
```

Constructors

```
class BaseClass()
{
public:
    BaseClass();
    //may also have other
    //constructors
private:
    //stuff here
}; //end BaseClass

BaseClass::BaseClass()
{
    //implementation here
}
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    //stuff here
}; //end DerivedClass

DerivedClass::DerivedClass()
{
    //implementation here
}
```

```
main()

DerivedClass my_derived_class;
//BaseClass default constructor called
//then DerivedClass constructor called
```

Constructors

```
class BaseClass()
{
public:
    BaseClass(int value);
    //stuff here

private:
    int base_member_;
};  //end BaseClass

 BaseClass::
 BaseClass(int value):
 base_member_(value)
 {
     //implementation here
 }

main()
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    //stuff here
};  //end DerivedClass

DerivedClass::DerivedClass()
{
    //implementation here
}
```

```
DerivedClass my_derived_class;
//PROBLEM!!! there is no default constructor to be called
//for BaseClass
```

Constructors

```
class BaseClass()
{
public:
    BaseClass(int value);
    //stuff here

private:
    int base_member_;
};  //end BaseClass

BaseClass::
BaseClass(int value):
base_member_(value)
{
    //implementation here
}

 main()
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    static const int INITIALIZATION_VAL = 0;
};  //end DerivedClass

DerivedClass::DerivedClass():
BaseClass(INITIALIZATION_VAL)
{
    //implementation here
}
```

Fix

```
DerivedClass my_derived_class;
// BaseClass constructor explicitly called by DerivedClass
//constructor
```

Destructors

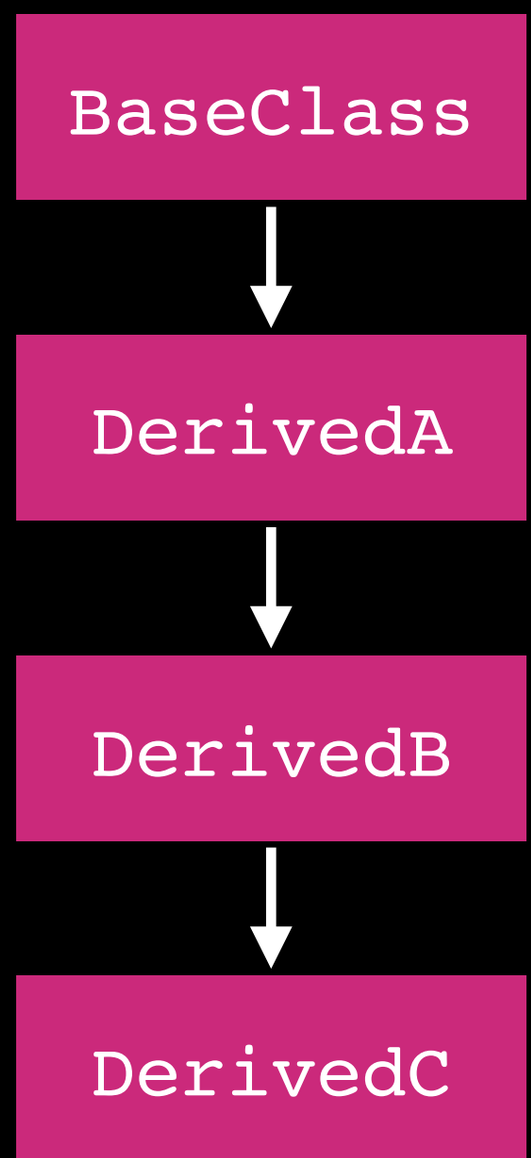
Destructor invoked if:

- program execution left scope containing object definition
- `delete` operator was called on object that was created dynamically

Destructors

Derived class destructor always causes base class destructor to be called implicitly

Derived class destructor is called before base class destructor



Order of calls to constructors when instantiating a DerivedC object:

```
BaseClass()
DerivedA()
DerivedB()
DerivedC()
```

Order of calls to destructors when instantiating a DerivedC object:

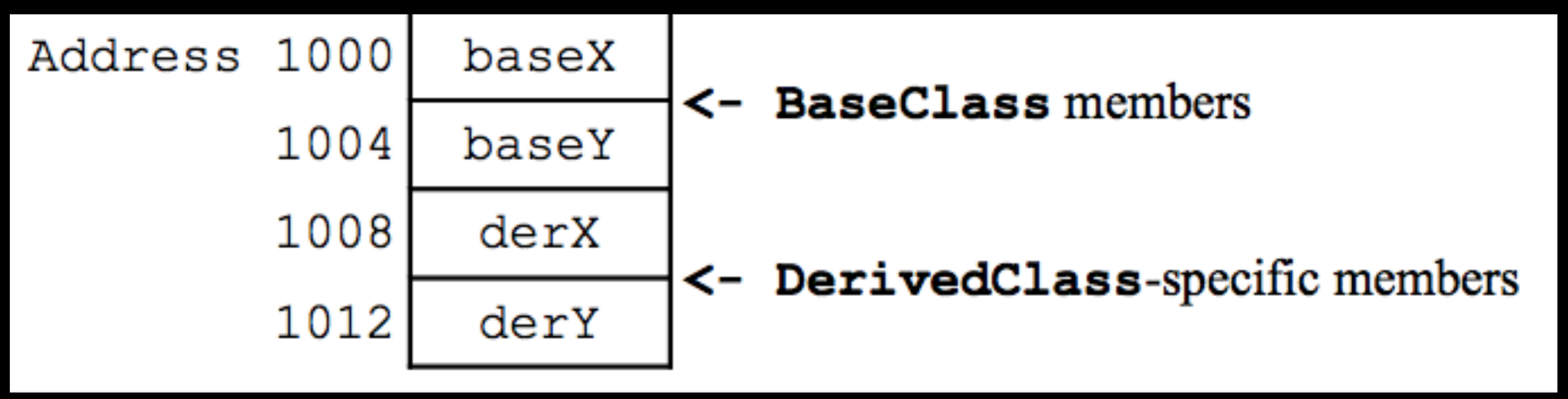
```
~DerivedC()
~DerivedB()
~DerivedA()
~BaseClass()
```

Basic Inheritance

No runtime cost

In memory `DerivedClass` is simply `BaseClass` with extra members tacked on the end

Basically saving to re-write `BaseClass` code



Abstract Class

A class that does not implement all or some of its members

Cannot be instantiated

Marked by: `virtual` *`method prototype`* `=0;`

Inheritance as Interface

Client interface: use templates and abstract classes to completely specify ADT (`BagInterface.h`)

Includes only public members of Bag

It is only an interface (no implementation)

Enforce information hiding

Client only needs interface to use ADT

Client cannot see private declaration

```
template<class ItemType>
class BagInterface
{
public:
    /** Gets the current number of entries in this bag.
    @return The integer number of entries currently in the bag. */
    virtual int getCurrentSize() const = 0;

    /** Checks whether this bag is empty.
    @return True if the bag is empty, or false
    if not. */
    virtual bool isEmpty() const = 0;

   /** Adds a new entry to this bag.
   @post  If successful, newEntry is stored in the bag
   and the count of items in the bag has increased by 1.
   @param newEntry  The object to be added as a new entry.
   @return  True if addition was successful, or false if not. */
   virtual bool add(const ItemType& newEntry) = 0;

    /** Removes one occurrence of a given entry from this bag, if possible.
    @post  If successful, anEntry has been removed from the bag
    and the count of items in the bag has decreased by 1.
    @param anEntry  The entry to be removed.
    @return  True if removal was successful, or false if not. */
    virtual bool remove(const ItemType& anEntry) = 0;
```

Means: “this method will not modify the object”

For now think of this as saying:
“I’m only declaring this but will not implement it.”
It will force derived class to implement it.

```
/** Removes all entries from this bag.
@post  Bag contains no items, and the count of items is 0. */
virtual void clear() = 0;

/** Counts the number of times a given entry appears in bag.
@param anEntry  The entry to be counted.
@return  The number of times anEntry appears in the bag. */
virtual int getFrequencyOf(const ItemType& anEntry) const = 0;


/** Tests whether this bag contains a given entry.
@param anEntry  The entry to locate.
@return  True if bag contains anEntry, or false otherwise. */
virtual bool contains(const ItemType& anEntry) const = 0;


/** Fills a vector with all entries that are in this bag.
@return  A vector containing all the entries in the bag. */
virtual std::vector<ItemType> toVector() const = 0;


}; // end BagInterface
```

Documenting Interface

Javadoc generates HTML documentation from Java source code. Your textbook uses it so I will show it to you

- @param
- @return
- ...

Also parameter names are in Java style, so we will keep them

- `anEntry` instead of `an_entry`

In general:

- be clear and thorough
- Interface comments are general and do not mention private data members -> no implementation detail

Programming Practice

Write two dummy classes `BaseClass` and `DerivedClass` (separate interface and implementation)

Play with constructors and destructors and try different scenarios (missing `BaseClass` constructor or only parameterized `BaseClass` constructor)

Test different types of inheritance and inspect behavior (eg. `public`, `private`, `protected`)

Try adding `AnotherDerivedClass`

Try overloading and overriding methods

`cout<<` from every member function including constructors and destructors to trace function calls and understand behavior