Inheritance
ADTs & Templates

Tiziana Ligorio

Today's Plan

Recap

Useful C++ / OOP

ADTs

Templates

Intro to Inheritance

Maybe More useful C++ / OOP

Announcements

- No drop-in office hours this week, email me if you need to make an appointment

Make sure you can successfully submit to Gradescope before the project due date

- Start early, work incrementally

Recap

OPP

- Abstraction
- Encapsulation
- Information Hiding

Classes

- Public Interface
- Private Implementation
- Constructors / Destructors

Interface

Implementation

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

SomeClass.hpp
(same as SomeClass.h)

```
#ifndef SOME_CLASS_H_
#define SOME_CLASS_H_

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


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 "SomeClass.hpp"

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


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


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


bool SomeClass::methodThree(int

someParameter)
{
     //implementation here
}
```

Some (Perhaps Review) Useful Concepts

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)
```

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:

```
Person(int id, string first = "", string last = "");

Person(143); // calls Person(143,"", "")
Person(143, “Gina”);   // calls Person(“143”,”Gina”, “”)
Person(423, “Nina”, “Moreno”);   // calls Person(423,“Nina”,”Moreno”)
```

Default Arguments

```
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)
```

Order Matters!
Parameters without default arguments must go first.

```
Animal(std::string name = "", bool domestic = false, bool predator = false);
```

IS DIFFERENT FROM

```
Animal(std::string name, bool domestic = false, bool predator = false);
```

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

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 member functions go here
        friend returnType someFriendFunction( parameter list);
    private:
        int some_data_member_;

 };                         // end SomeClass
```

IMPLEMENTATION (SomeClass.cpp):

Not a member function

```
            returnType someFriendFunction( parameter list)
            {
                // implementation here
                some_data_member_ = 35; //has access to private data
            }
```

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_) &&
             (object1.memberB_ == object2.memberB_) && … );
}
```

Enum

A user defined datatype that consist of integral constants

Type name (like `int`)

Possible values: like 0,1, 2, …

Why? Readability

```
enum season {SPRING, SUMMER, AUTUMN, WINTER };

enum animal_type {MAMMAL, FISH, BIRD};
```

By default = 0, 1, 2, …

To change default:

```
enum ta_role {MAMMAL = 5, FISH = 10, BIRD = 20};
```

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

Basic Inheritance

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

    void setPaperSize(int size);
    void setOrientation(const string& orientation);
    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
```

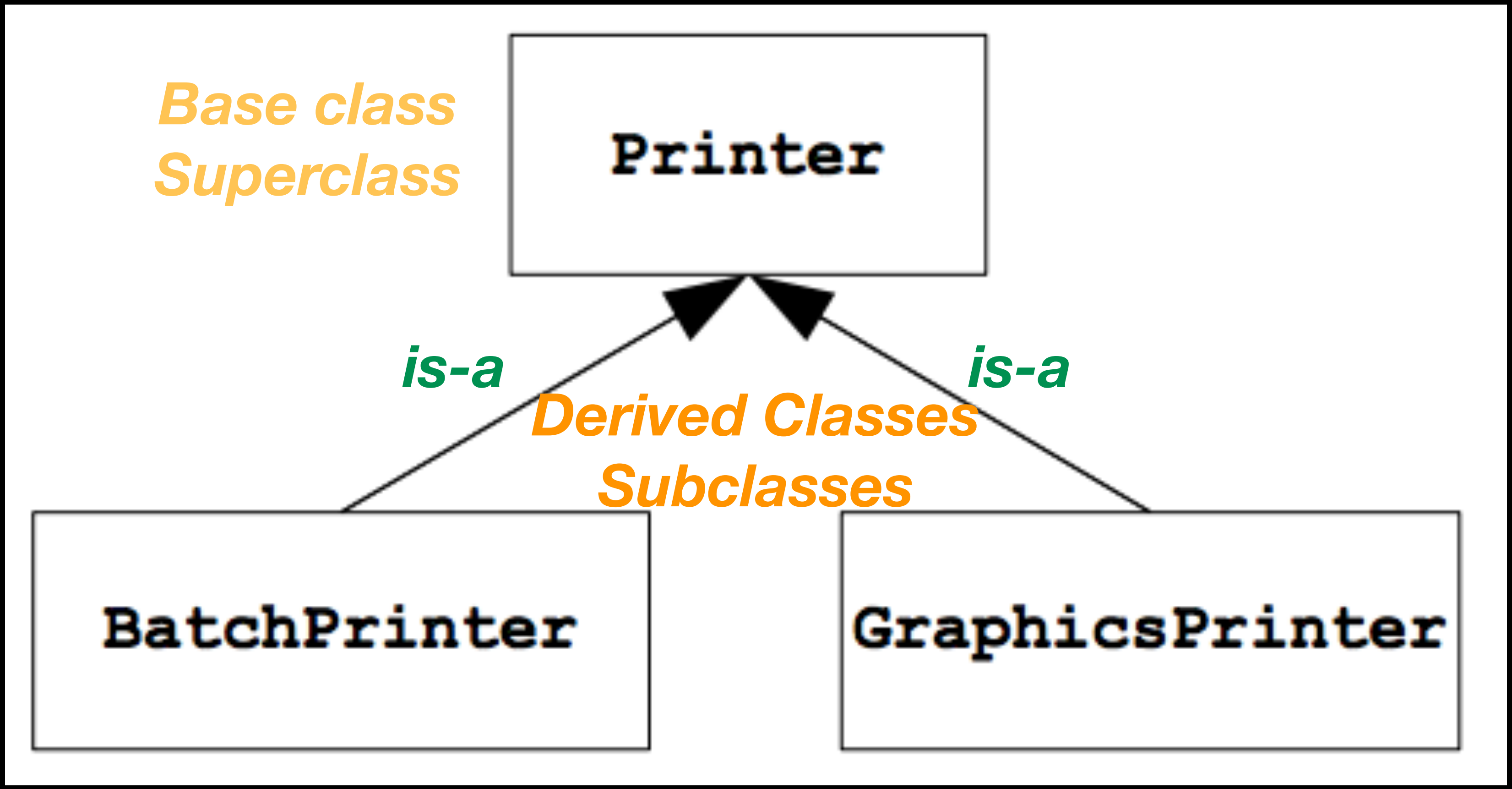
Basic Inheritance

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

    void setPaperSize(int size);
    void setOrientation(const string& orientation);
    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(){ }
```

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

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

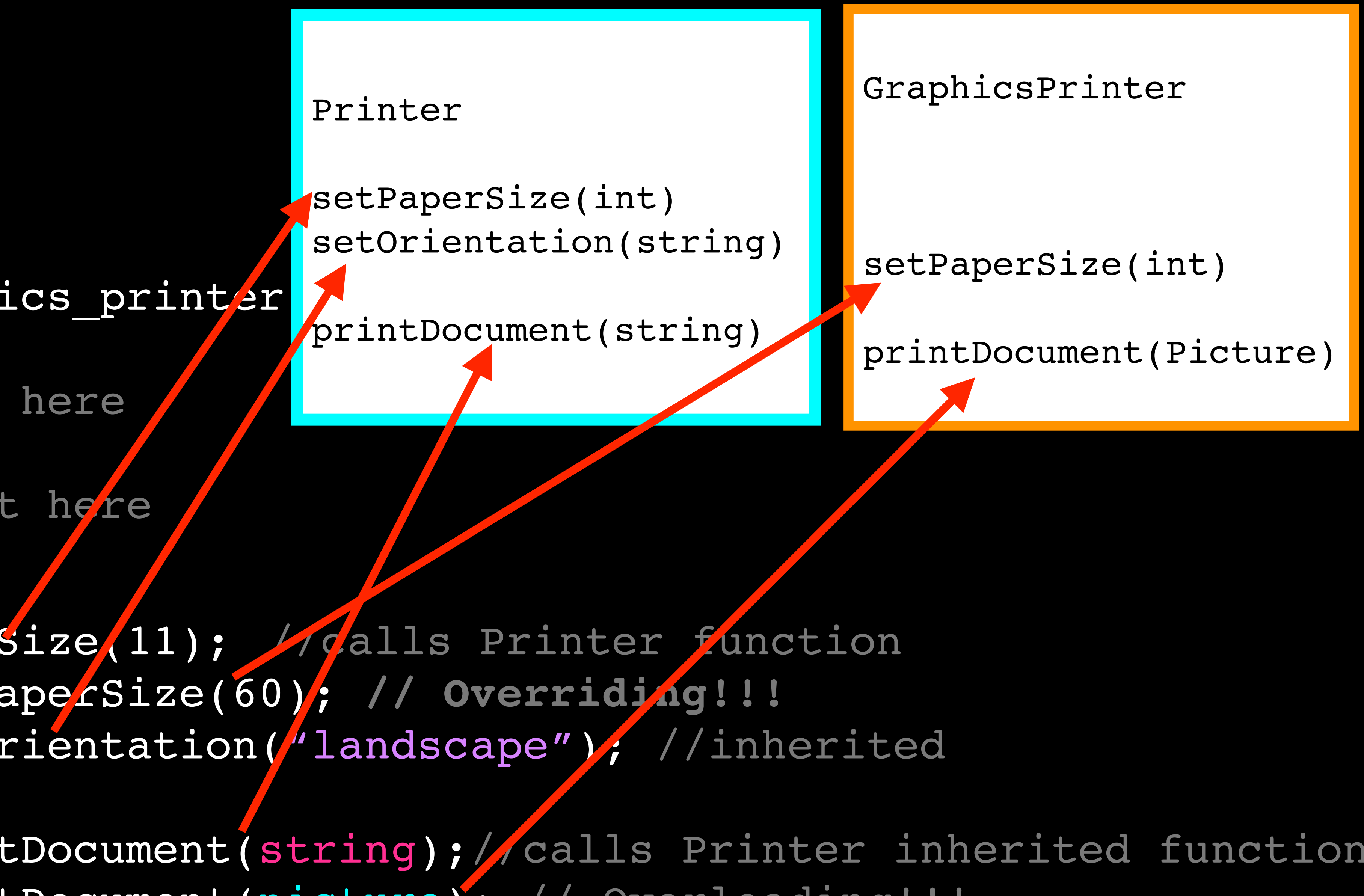
```cpp
class GraphicsPrinter: public Printer    // inherit from printer
{
public:
    //Constructor, destructor
    void setPaperSize(const int size);
    void printDocument(const Picture& picture);//some Picture object


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

Overrides `setPaperSize()`

Overloads `printDocument()`

main()

```
Printer base_printer;
GraphicsPrinter graphics_printer
Picture picture;
// initialize picture here
string document;
// initialize document here


base_printer.setPaperSize(11);  //calls Printer function
graphics_printer.setPaperSize(60); // Overriding!!!
graphics_printer.setOrientation("landscape"); //inherited

graphics_printer.printDocument(string);//calls Printer inherited function
graphics_printer.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 direct access to base class `private` members. However, can call public functions of the base class, which in turn do have access base classe's 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

INTERFACE

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

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

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

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

IMPLEMENTATION

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

main()

```
DerivedClass my_derived_class;
//BaseClass compiler-supplied default constructor called
//then DerivedClass constructor called
```

Constructors

INTERFACE

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

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

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

IMPLEMENTATION

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

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

main()

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

Constructors

INTERFACE

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

    private:
        int base_member_;
    }; //end BaseClass
```

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

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

IMPLEMENTATION

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

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

main()

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

Constructors

INTERFACE

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

private:
    int base_member_;
}; //end BaseClass
```

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

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

IMPLEMENTATION

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

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

Fix

main()

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

Destructors

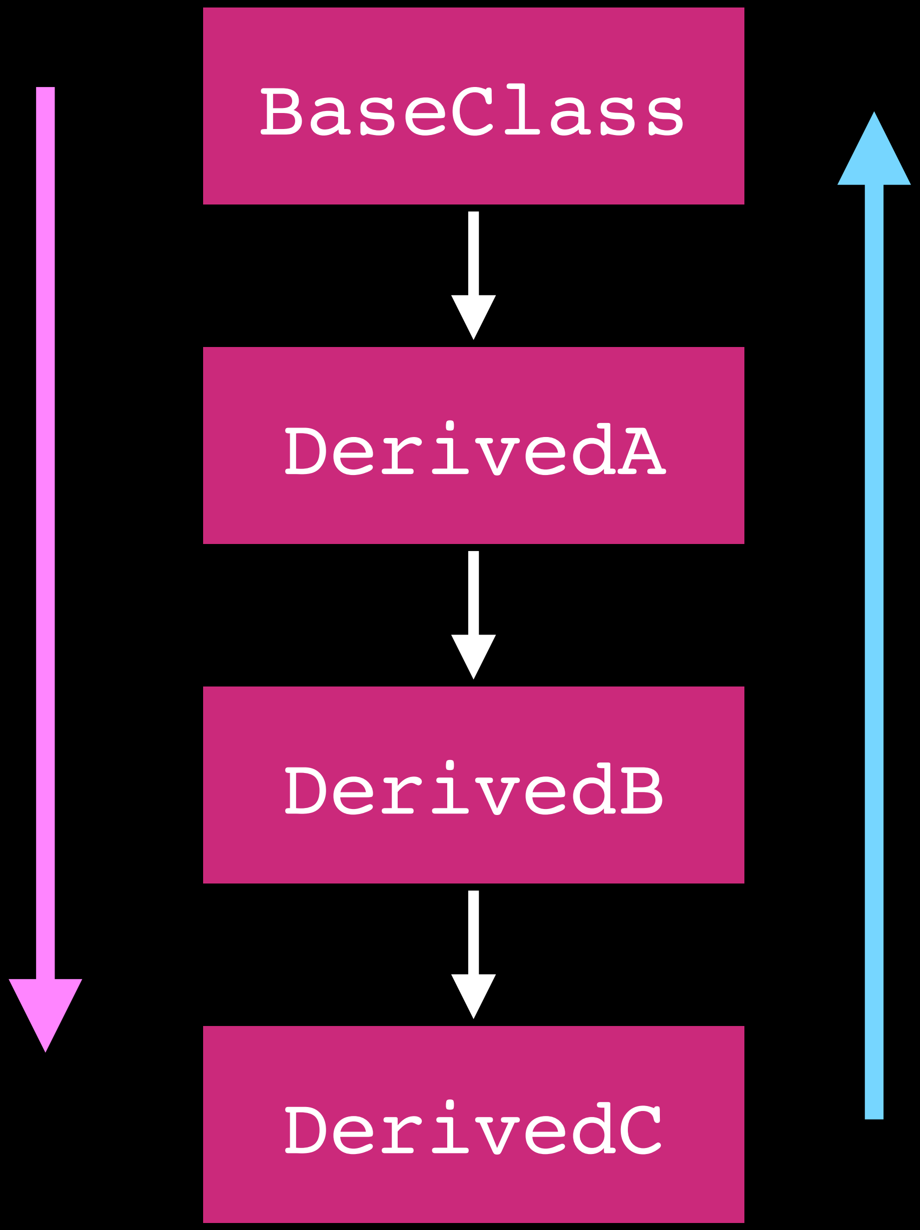
Destructor invoked if:

- 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()
```

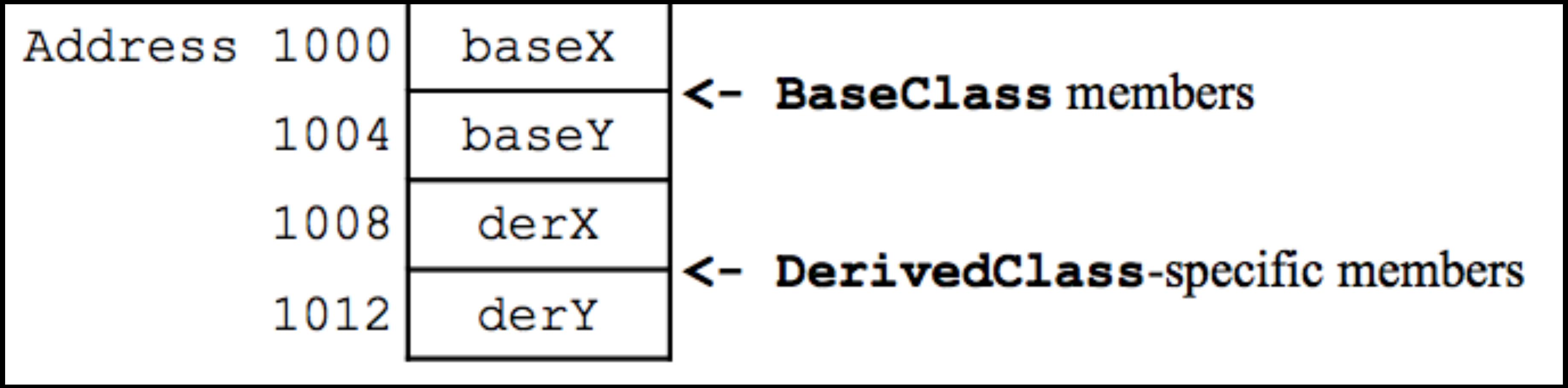
Order of calls to destructors when instantiating a DerivedC object:

```
~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 Data Type

Data and Abstraction

Operations on data are central to most solutions

Think abstractly about data and its management

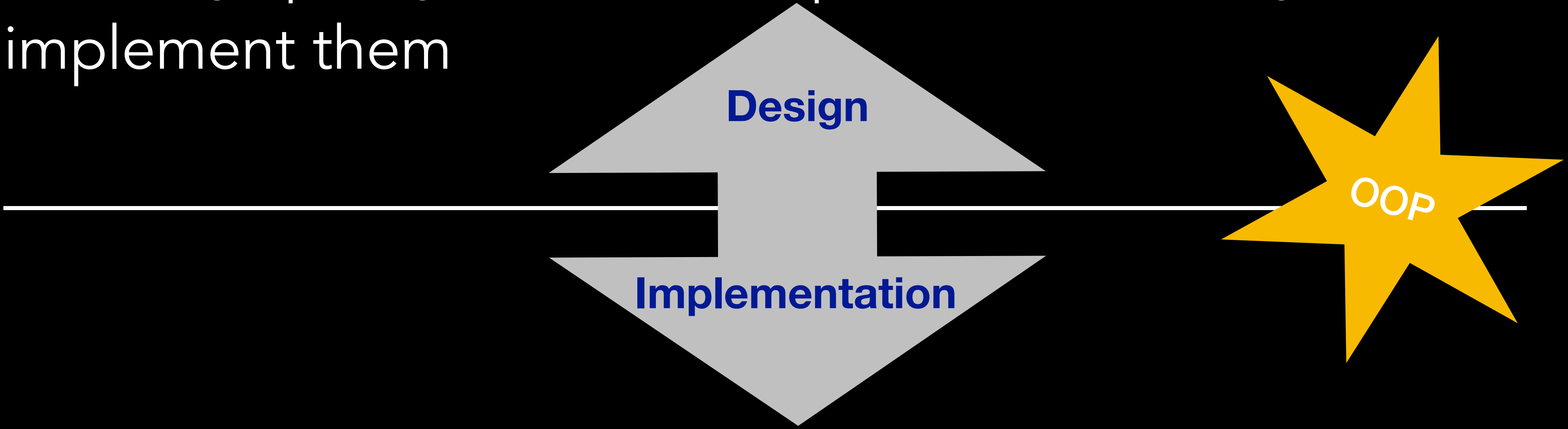
Typically need to

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

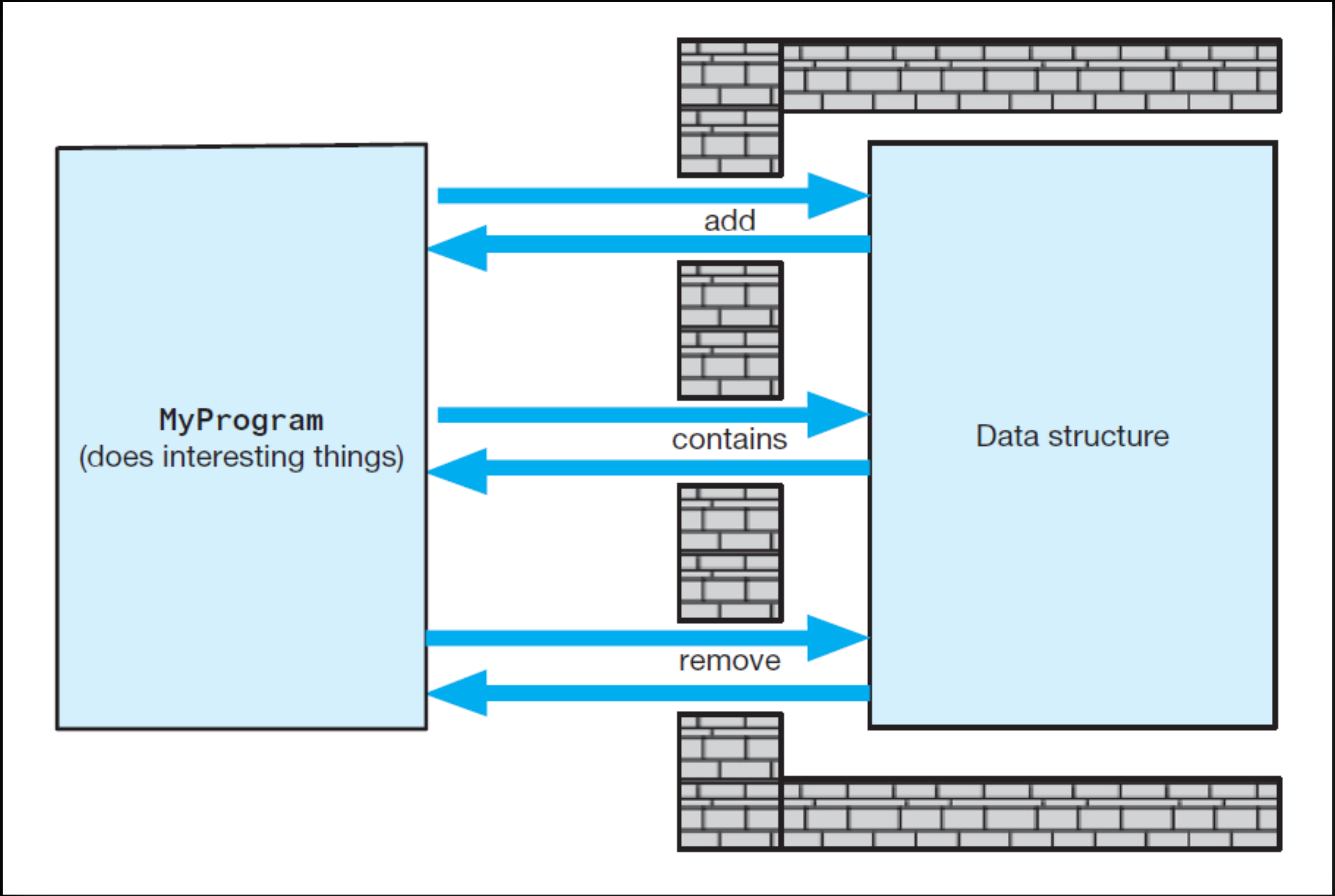
Abstract Data Type

A collection of data (container) and a set of operations on the data

Carefully specify and ADT's operations before you implement them



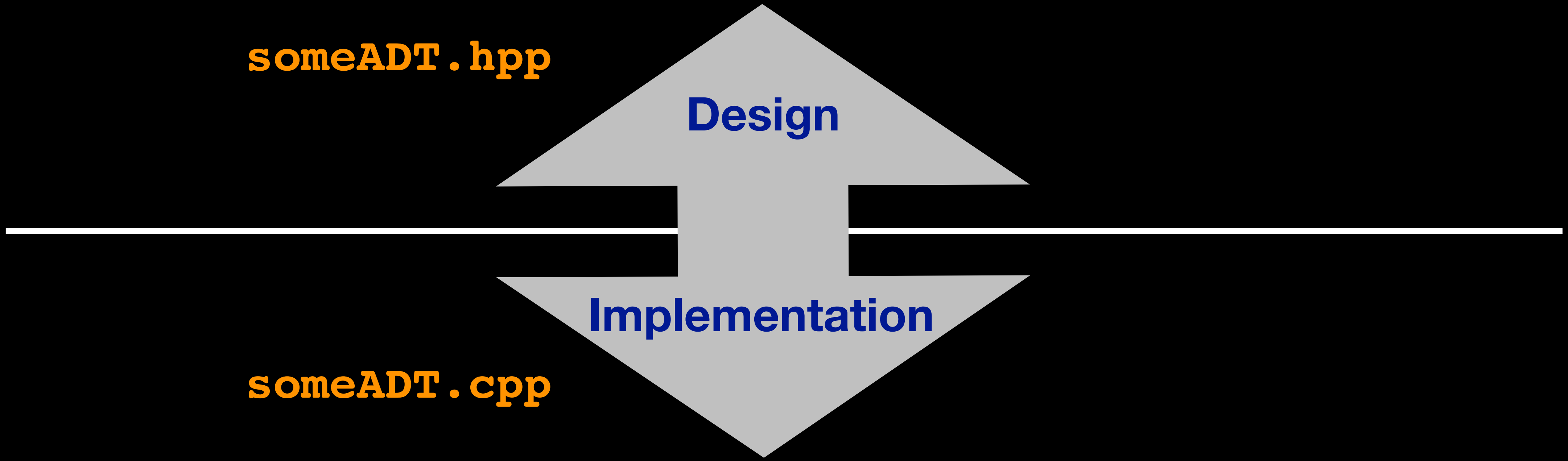
In C++ member variables and member functions implement the Abstract Data Type


MyProgram
(does interesting things)
add
contains
remove
Data structure

Class

```
class someADT
{
    access_specifier    // can be private, public or protected
    data_members        // variables used in class
    member_functions    // methods to access data members

 };                     // end someClass
```

someADT.hpp



someADT.cpp

Designing an ADT

- What data does the problem require?
    - Data
    - Organization
- What operations are necessary on that data?
    - Initialize
    - Display
    - Calculations
    - Add
    - Remove
    - Change

Throughout the semester we will consider several ADTs

Let’s start from the simplest possible!

Design the Bag ADT

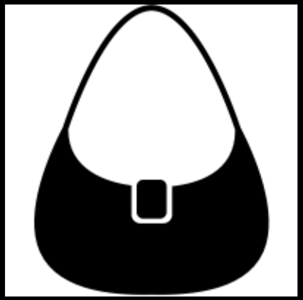
Contains things

Container or Collection of Objects

Objects are of same type

No particular order

Can contain duplicates



Lecture Activity

Design step 1 — Identify Behaviors

Bag Operations:

1.
2.
3.
4.
5.
6.

…

Design step 1: Identify Behaviors

Bag Operations:

1. Add an object to the bag
2. Remove an occurrence of a specific object form the bag if it's there
3. Get the number of items currently in the bag
4. Check if the bag is empty
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++ Standard Template Library)

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

e.g.

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

In this course cannot use STL or particular functions 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 T> // this is a template definition
class Bag
{

   //class declaration here


};
#include “Bag.cpp”    ← Explained next
#endif //BAG_H_
```

Declaration

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

    //class declaration

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

The book uses `ItemType`
I'm going to change it to `T` which is often used

`class` here could be replaced by `typename`
often interchangeable but can make a difference in some cases, we will not go into the details here

for this course we will use `class`

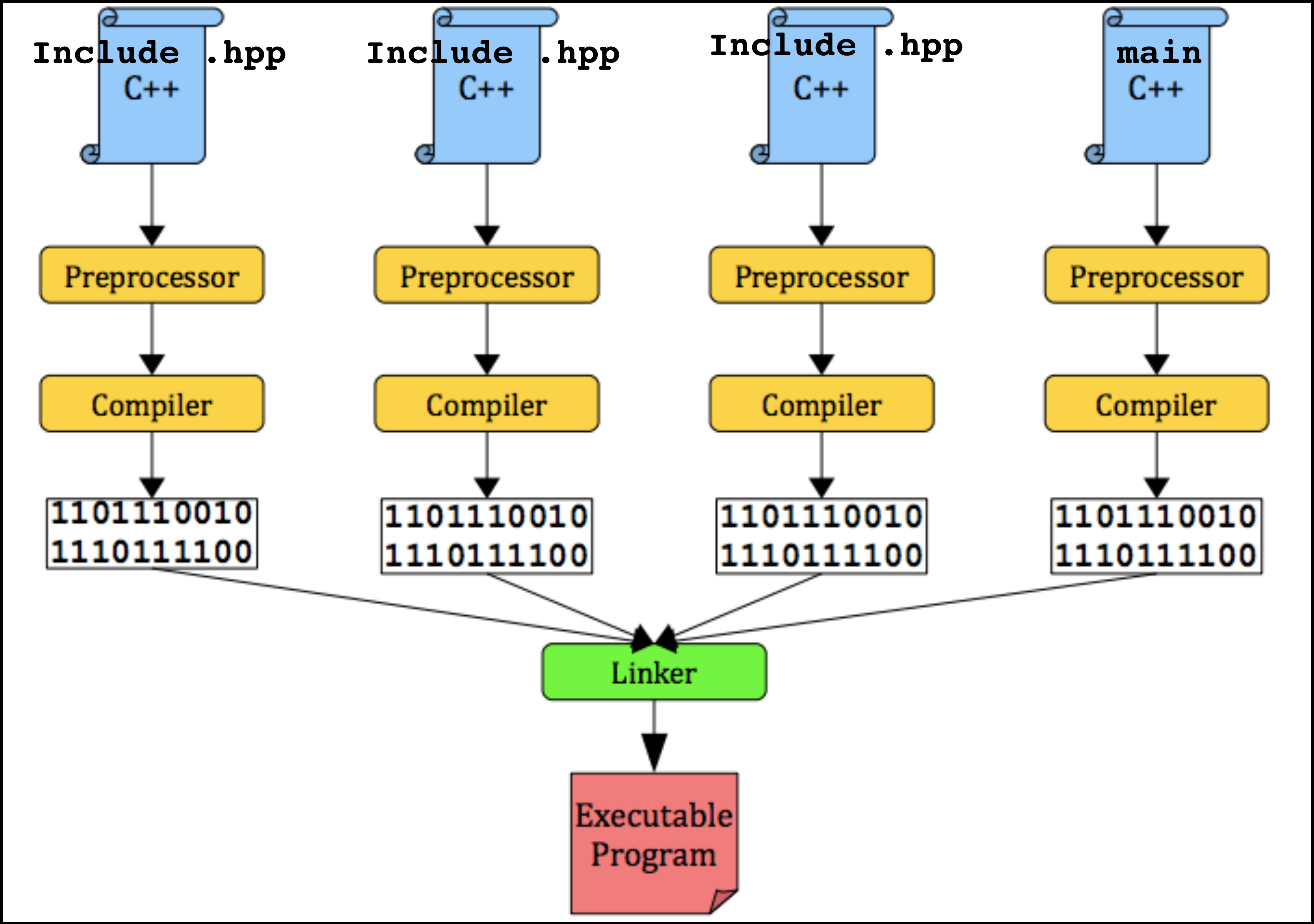
Implementation

```
#include “Bag.hpp”

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


    //more member function implementation here
```

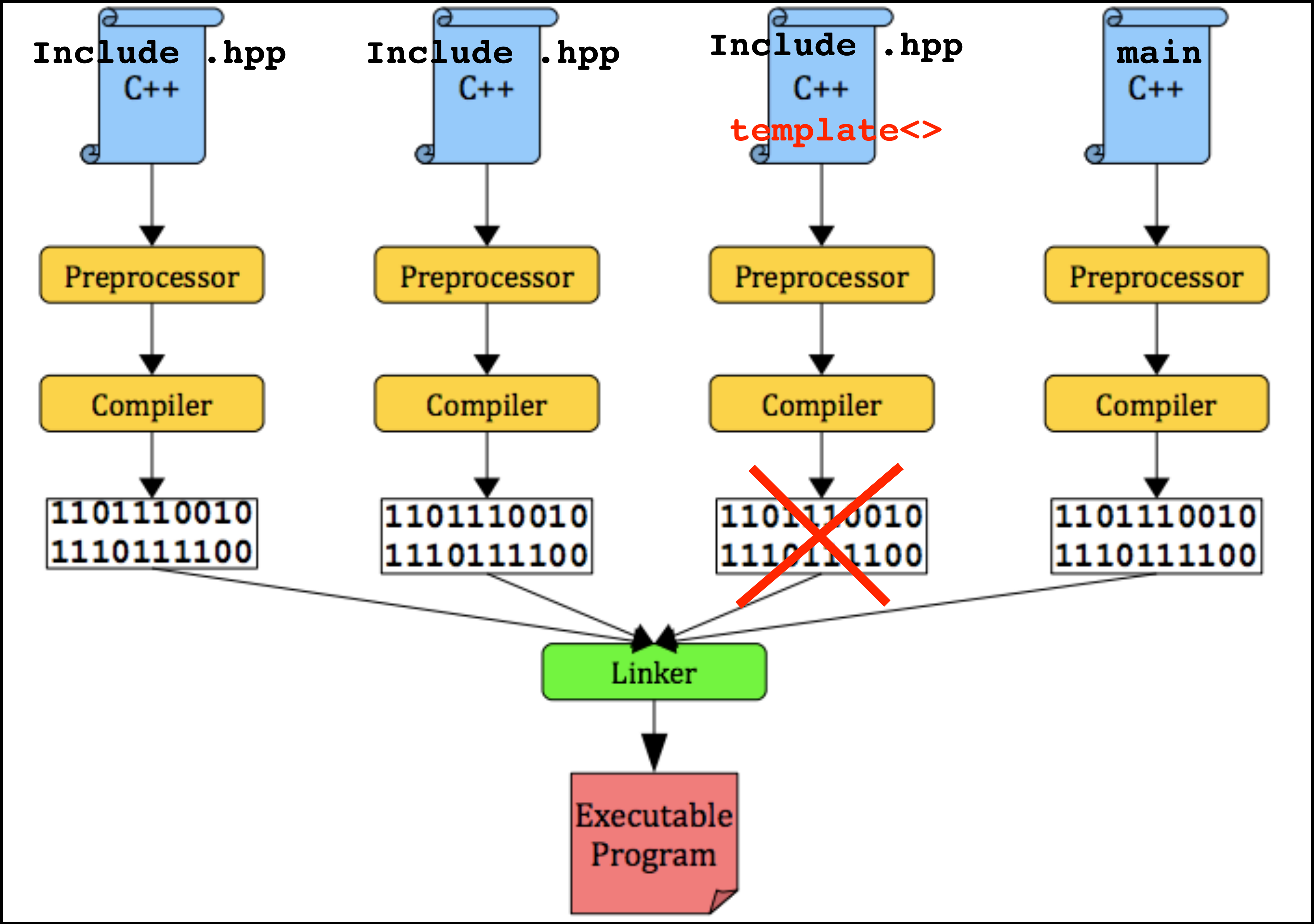
Instantiation

```
#include “Bag.hpp”

int main()
{

    Bag<string> string_bag;
    Bag<int> int_bag;
    Bag<someObject> some_object_bag;

    std::vector<int> numbers;
    //stuff here

    return 0;

}; // end main
```

Separate Compilation



Linking with Templates



Linking with Templates

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

```
#ifndef MYTEMPLATE_H_
#define MYTEMPLATE_H_
template<class T>
class MyTemplate
{

//stuff here


} //end MyTemplate
#include "MyTemplate.cpp"
#endif  //MYTEMPLATE_H_
```

IMPORTANT

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

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

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

NOT `g++ -o my_program MyTemplate.cpp main.cpp`

Lecture Activity

```
template<class T> // this is a template definition
class MyTemplate
{
    void setData(T some_data); //mutator
    T getData() const; //accessor

private:
    T my_data_; //this is the only private data member

}
```

Write a `main()` function that instantiates 3 different `MyTemplate` objects with different types (e.g. `int`, `string`, `bool`) and makes calls to their member functions and show the output. E.g:

```
    MyTemplate<double> double_object;
    double_object.setData(3.0);
    cout << double_object.getData() << endl; // outputs 3.0
```

Try It At Home

Write a dummy `MyTemplate` interface and implementation
(`MyTemplate.hpp, MyTemplate.cpp`)
Test it in `main()`
Make sure you can compile a templated class
(REMEMBER YOU DON'T COMPILE IT!!!)
YOU WILL THANK ME

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

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


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

    /** Removes one occurrence of a given entry from this bag, if possible.
    @post  If successful, an_entry has been removed from the bag
    and the count of items in the bag has decreased by 1.
    @param an_entry  The entry to be removed.
    @return  True if removal was successful, or false i
    bool remove(const T& an_entry);
```

Means: “this method will not modify the object”

Means: “this method will not modify the parameter”

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

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


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


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


}; // end BagInterface
```

Recap

We designed a Bag ADT by defining the operations on the data

We templatized it so we can store any data type

NEXT: Implementation