

Abstract Data Types & Templates

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

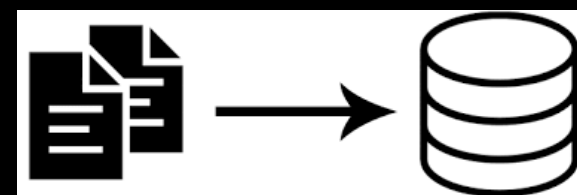
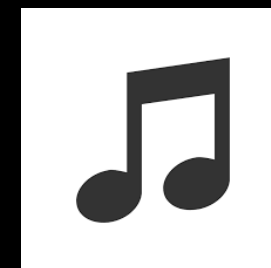
Add data

Remove data

Retrieve

Ask questions about data

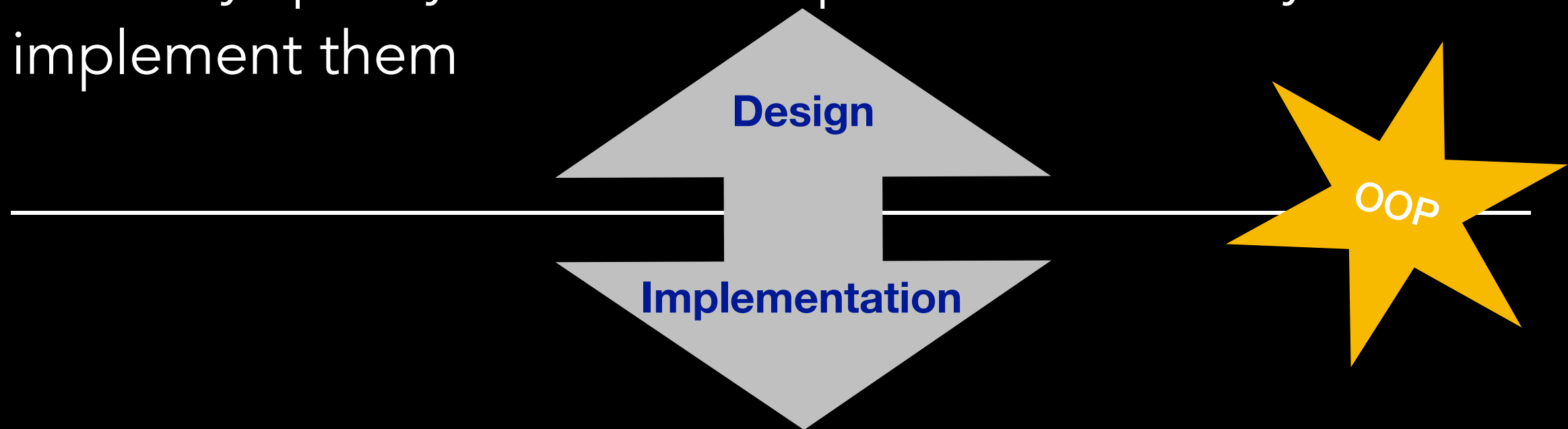
Modify data



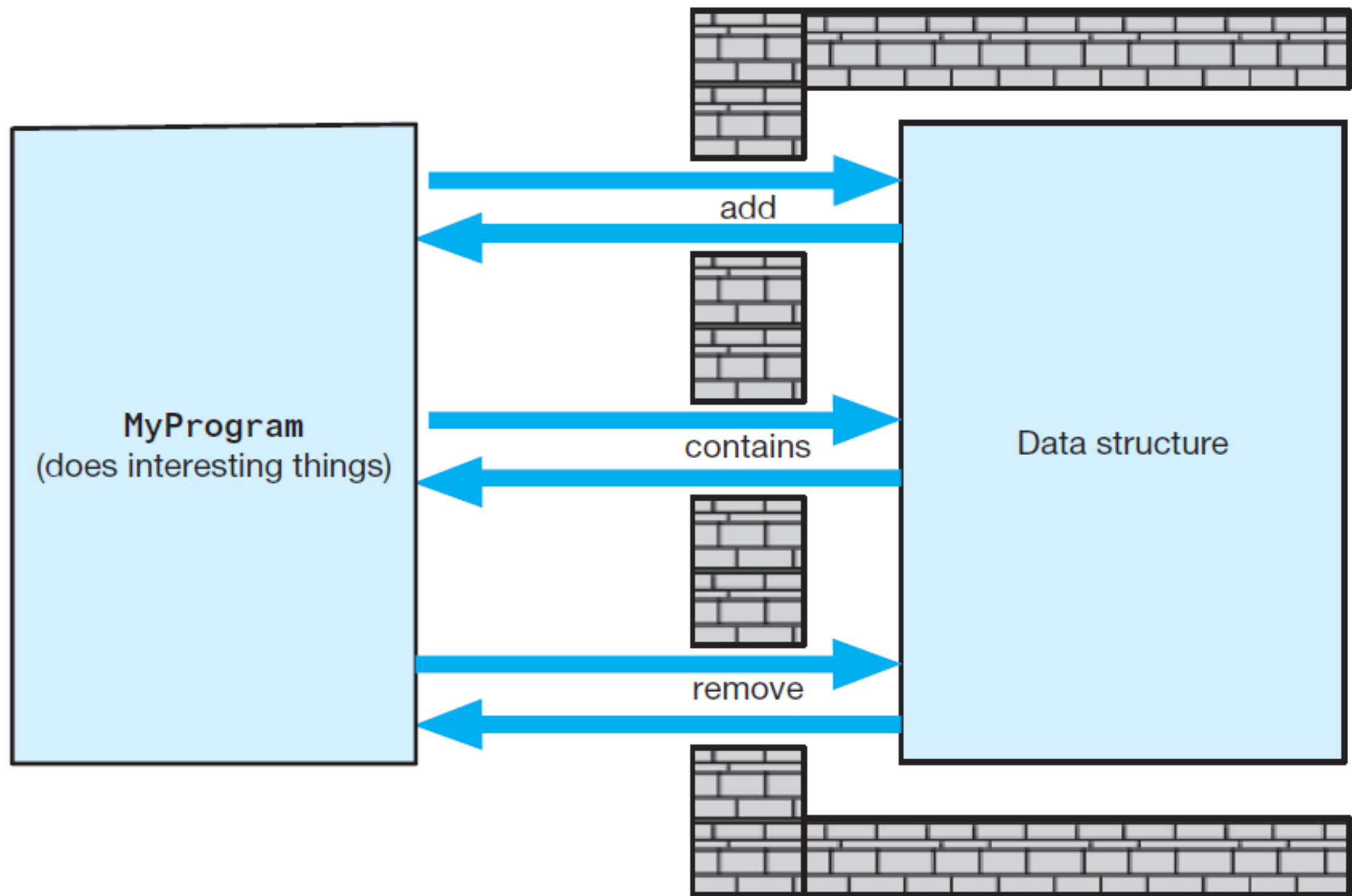
Abstract Data Type

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



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

Design

Implementation

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

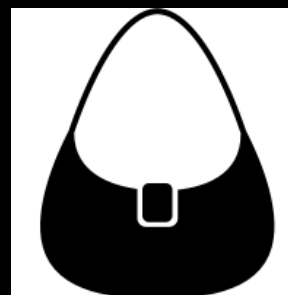
Objects are of same type



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 from 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 or 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>
```

```
...
```

```
vector<type> vector_name;
```

e.g.

```
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<typename ItemType>
class Bag // this is a template definition
{

    //class declaration here

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

Declaration

```
#ifndef BAG_H_
#define BAG_H_
template<typename ItemType> // this is a template definition
class Bag
{
    //class declaration here
};
#include "Bag.cpp"
#endif //BAG_H_
```

Explained next

Sometimes **T** is used instead of
ItemType

typename here could be replaced
by **class**
often interchangeable but can make

Implementation

```
#include "Bag.hpp"
```

```
template<typename ItemType>
```

```
bool Bag<ItemType>::add(const ItemType& new_entry){
```

```
    //implementation here
```

```
}
```

```
    //more member function implementation here
```

Instantiation

```
#include "Bag.hpp"
```

```
int main()  
{
```

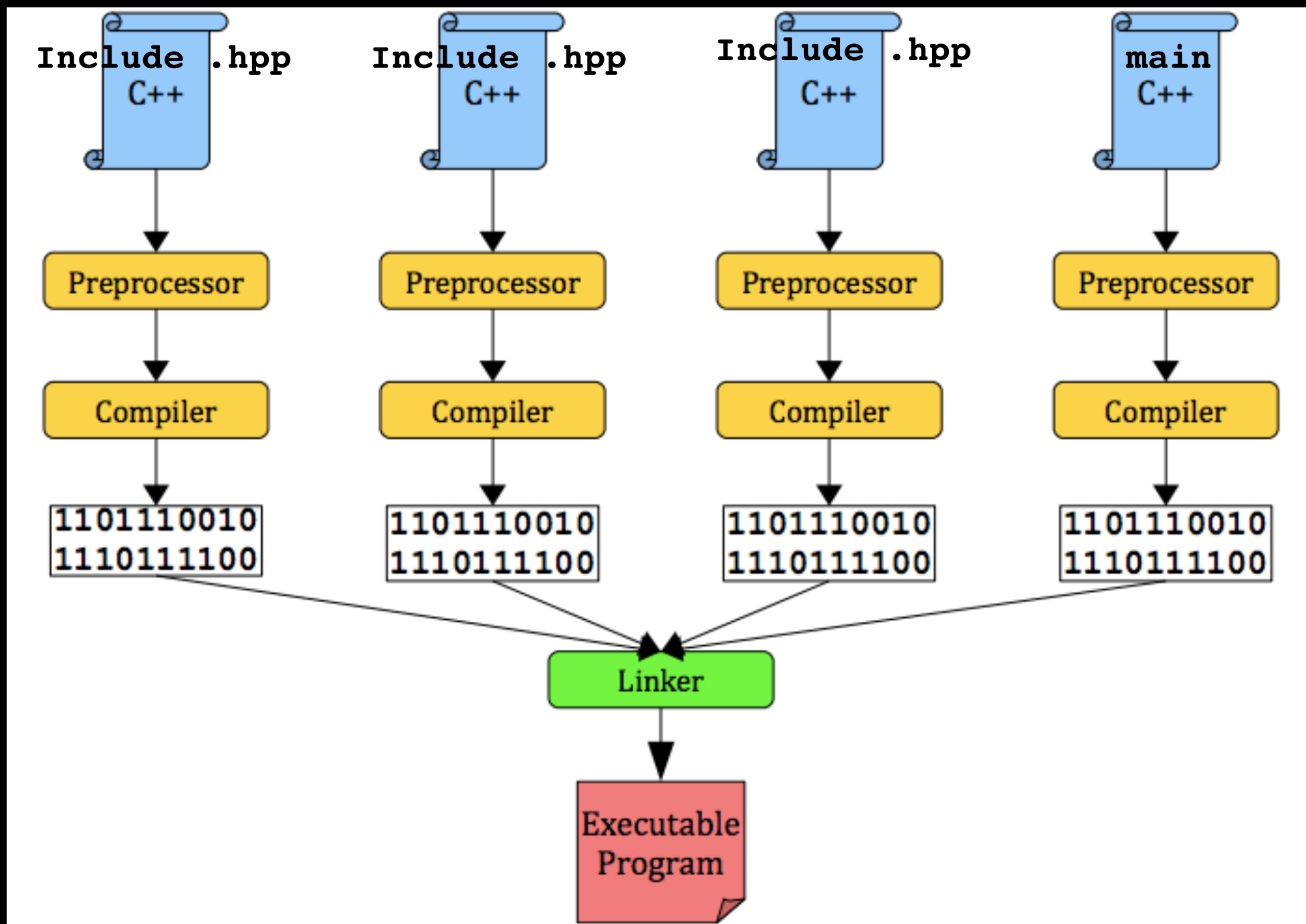
```
    Bag<string> string_bag;  
    Bag<int> int_bag;  
    Bag<someObject> some_object_bag;
```

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

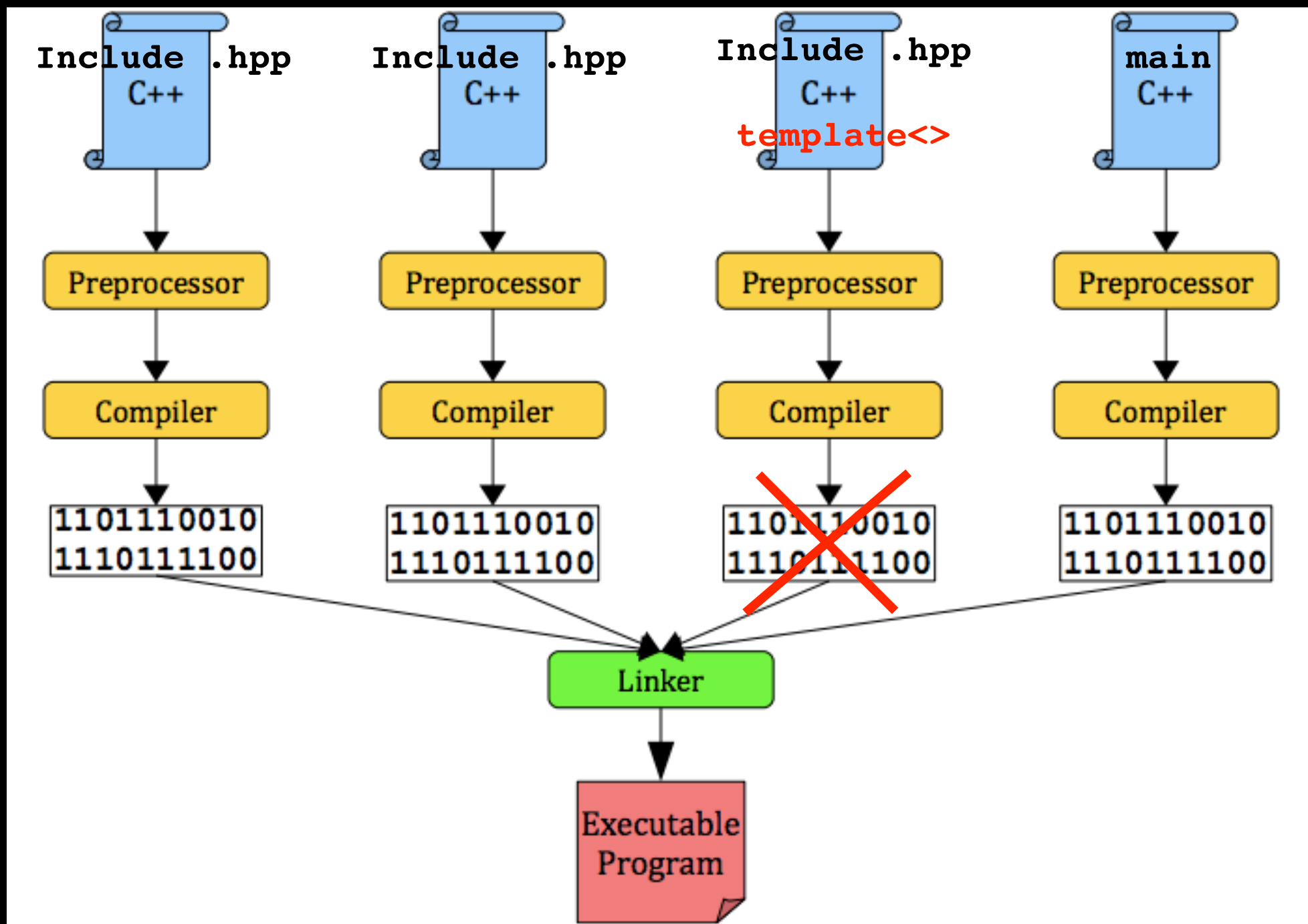
```
    return 0;
```

```
}; // 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<typename ItemType>
class MyTemplate
{

//stuff here

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



**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<typename ItemType> //this is a template definition
class MyTemplate
{
public:
    void setData(ItemType some_data); //mutator
    ItemType getData() const; //accessor

private:
    ItemType 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<typename ItemType>
```

```
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 ItemType& 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 if not. */
```

```
    bool remove(const ItemType& 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 ItemType& 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 ItemType& 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<ItemType> toVector() const;

}; // end BagInterface

```

Recap

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

We templated it so we can store any data type

NEXT: Implementation