Lecture 1

COURSE INTRODUCTION, ADT BAG

Overview

- Welcome to ECE 302 Data Structures and Algorithms
- Canvas organization and walk-through
- Tentative 6 weeks schedule is also posted on Canvas, with all the reading.
  - You can also find all the TA emails here….
- Textbook is to be available via Canvas.
  - Carrano and Henry, Data Abstraction & Problem Solving with Walls and Mirrors C++
- We will spend some time today getting a VM installed.

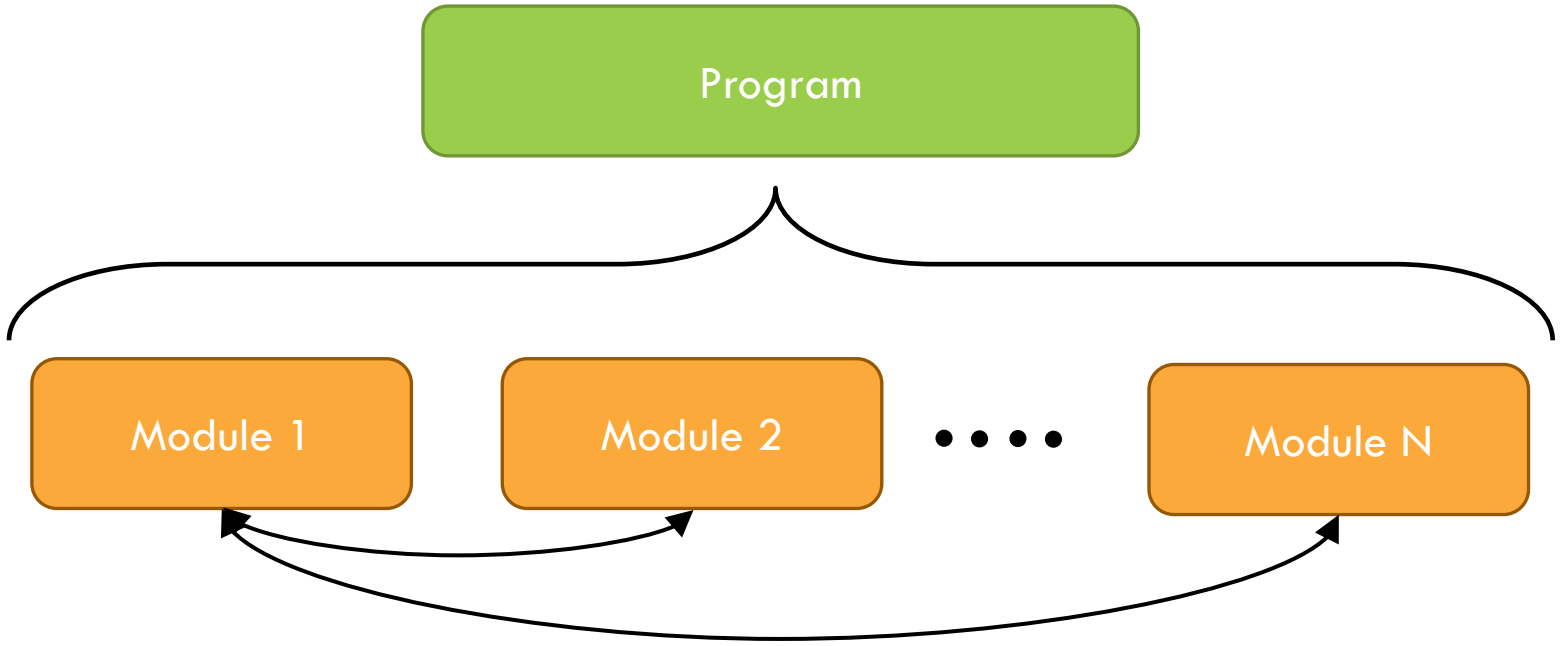
An expression of a computational method in a computer language is a program

```
// simple C++ program
#include <iostream>
#include <cstdlib>

int main()
{
   std::cout << "Hello World!”<< std::endl;
   return EXIT_SUCCESS;
};
```

Programs are made up of modules

- **Modules** are self contained units of code that solve sub-problems of the larger programming task.



Object-Oriented Concepts and Design

- Object-oriented analysis and design (OOAD)
  - Process for solving problems
- Solution
  - Computer program consisting of system of interacting classes of objects
- Object
  - Has set of characteristics, behaviors related to solution
- Requirements of a solution
  - What solution must be, do
- Object-oriented design
  - Describe solution to problem
  - Express solution in terms of software objects
  - Create one or more models of solution

Aspects of Object-Oriented Solution

Principles of object-oriented programming

- Encapsulation: Objects combine data and operations.
- Inheritance: Classes inherit properties from other classes.
- Polymorphism: Objects determine appropriate operations at execution time.

Cohesion

- Each module should perform one well-defined task
- Benefits
  - Well named, self-documenting
  - Easy to reuse
  - Easier to maintain
  - More robust

Coupling

- Measure of dependence among modules
- Dependence
  - Sharing data structures or calling each other's methods
- Modules should be loosely coupled
  - Highly coupled modules should be avoided

- Benefits of loose coupling in a system
  - More adaptable to change
  - Easier to understand
  - Increases reusability
  - Has increased cohesion

This course is about Abstract Data Types (ADT) and Algorithms

- Abstraction separates the purpose of program modules from the implementation details.
- An algorithm is a finite set of rules that give a sequence of operations for solving a specific type of problem.

Abstraction separates the purpose of program modules from the implementation details.

- One way to represent a list in C++ (initially empty)

```
typedef char itemtype;

const unsigned int LEN = 256;

itemtype *mylist;

unsigned int mylistlen= 0;
mylist= new itemtype[LEN+1];
*mylist = 0;
```

Abstraction separates the purpose of program modules from the implementation details.

- One way to append to that list

```
typedef char itemtype;

const unsigned int LEN = 256;

itemtype *mylist;

unsigned int mylistlen= 0;
mylist= new itemtype[LEN+1];
*mylist = 0;
```

```
assert(mylistlen < LEN);

*(mylist + mylistlen) = A;
*(mylist + mylistlen + 1) = 0;
mylistlen++;
```

We can define a list based solely on its behavior.

- A list is a sequence of (possibly ordered) items with a beginning and an end. The operation append places an item in the list.
- There is no mention of how the list is implemented.
- This is an abstract data type (ADT).

```
List<char> MyList;
MyList.append(A);
```

Why do we care?

```
typedef char itemtype;

const unsigned int LEN = 256;

itemtype *mylist;

unsigned int mylistlen= 0;
mylist= new itemtype[LEN+1];
*mylist = 0;

// ------------------------

assert(mylistlen < LEN);

*(mylist + mylistlen) = A;
*(mylist + mylistlen + 1) = 0;
mylistlen++;
```

```
List<char> MyList;
MyList.append(A);
```

Which code base do **you** want to:

- Read?
- Maintain?
- Modify?

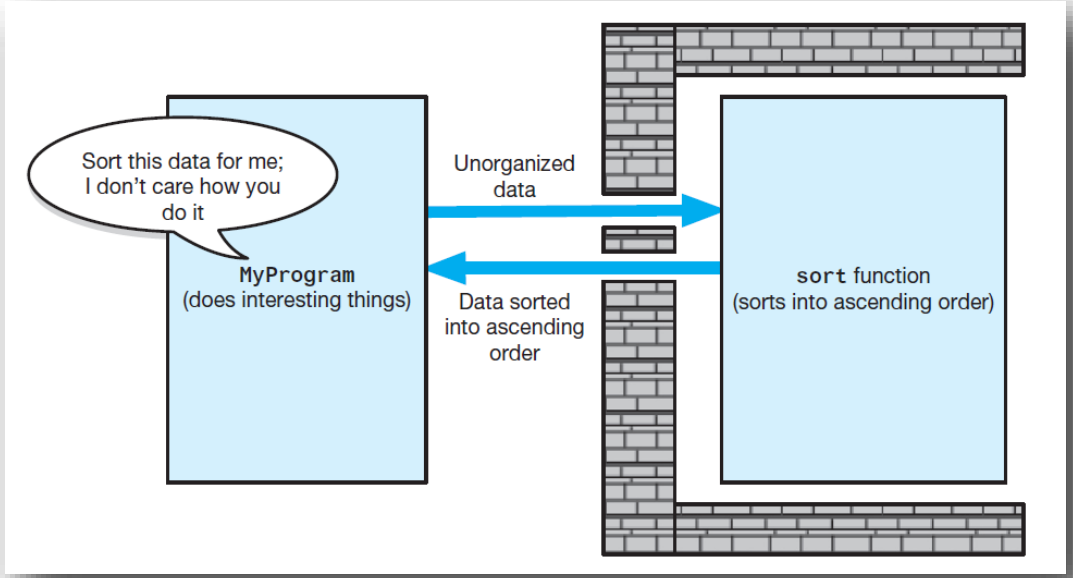
- Abstraction is necessary because of how we **think**

Abstraction

- Separate purpose of a module from its implementation
- Specifications do not indicate how to implement
  - Able to use without knowing implementation

Information Hiding

- Abstraction helps identify details that should be hidden from public view
  - Ensured no other module can tamper with these hidden details.
- Isolation of the modules cannot be total, however
  - Client must know what tasks can be done, how to initiate a task

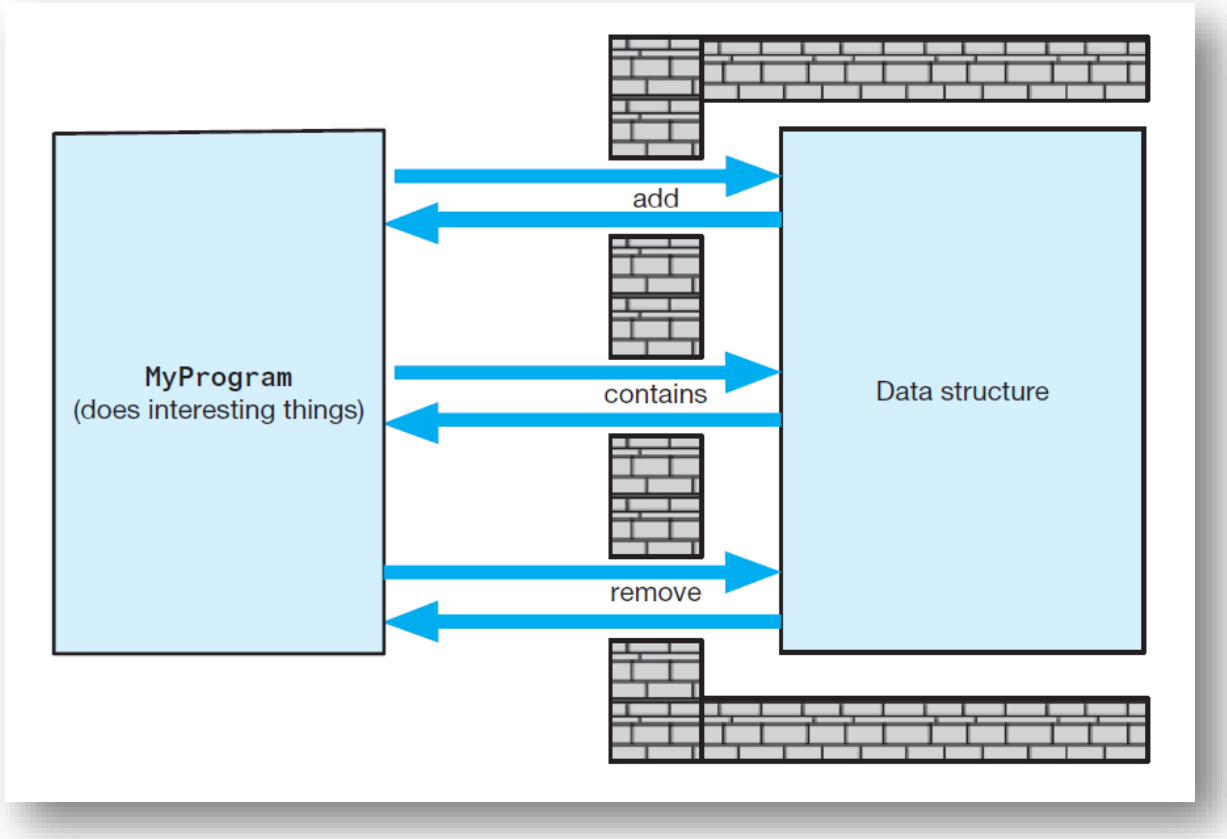


Minimal and Complete Interfaces

- Interface for a class made up of publicly accessible methods and data
- Complete interface for a class
  - Allows programmer to accomplish any reasonable task
- Minimal interface for a class
  - Contains method if and only if that method is essential to class's responsibilities

Abstract Data Types (ADT)

- Typical operations on data
  - Add data to a data collection.
  - Remove data from a data collection.
  - Ask questions about the data in a data collection.
- An ADT: a collection of data *and* a set of operations on data
- A data structure: an implementation of an ADT within a programming language



Designing an ADT

- Evolves naturally during the problem-solving process
  - What data does a problem require?
  - What operations does a problem require?
- ADTs typically have initialization and destruction operations
  - Assumed but not specified at this stage
- You can use an ADT to implement another ADT
  - Example: Date-Time objects available in C++ for use in various contexts
  - Possible to create your own fraction object

$$\left\{ \frac{a}{b} \mid a,\ \mathrm{b} \in \text{Integers},\ b \neq 0 \right\}$$

to use in some other object which required fractions

The ADT Bag

- Consider a virtual "bag" as an abstract data type.
  - We are specifying an abstraction inspired by an actual physical bag
  - Basic function to contain its items
  - Can be unordered and possibly contain duplicate objects
  - We insist objects be of same or similar types
- Knowing just its interface
  - Can use ADT bag in a program

`construct()`: construct and empty bag
`destroy()`: destroy the bag and any contents
`add( Item )`: add an `Item` to the bag, returns true on success, else false
`remove( Item )`: remove a single instance of `Item` from the bag, returns true on success, else false
`isEmpty()`: returns true if the bag has no contents, else false
`getCurrentSize()`: returns the number of items in the bag as an integer
`clear()`: removes all items in the bag
`getFrequencyOf( Item )`: the number of times `Item` appears in the bag
`contains( Item )`: returns true if at least one `Item` is in the bag, else false

Classes are the primary mechanism for implementing ADTs in C++

- A type is a concrete representation of a concept.
  - Type float approximates a real number
- A class is a user-defined type that extends the built-in types
  - Class Bag models the concept of the ADT Bag.
- Classes provide many advantages for implementing ADTs.
  - They can hide implementation details via private.
  - They provide a means of forcing the ADT interface to be used.
  - They enable type-checking on complex concepts.
  - They assist with assertion checking and maintaining constraints.
- All of this helps to keep the object (and instance of the type) in a well-defined state

Templates

- What is the difference between a bag of strings and a bag of integers?
  - The data type
- Really, there isn't a big difference in how they should be implemented
  - Can they be copied?
  - Can they be moved?
  - Can they be tested for equivalence?
- We can declare a generic class (template)

```
template<typename T> class Bag;
```

Generics in C++

- Templates
  - Elevate types to be generic – named but unspecified
  - Promotes code reuse
    - Types must meet the functionality required by the template
  - Helps you to understand and use template packages such as C++ STL
    - STL: Standard Template Library

Example 1: template function to swap

- A simple example is a function to swap the contents of two variables
  - Similar to `std::swap`

```
template< typename T >
void swap(T& a, T& b)
{
  T temp(b);
  b = a;
  a = temp;
}
```

Example 1: template function to swap

- The symbol `T` acts like a variable, in fact it is a *type* variable.
  - Defined this way swap is generic, I can use it on any type that can be copied.
  - For example:

```
int a = 1;
int b = 2;

std::cout << a << ", " << b << std::endl;
swap(a,b);
std::cout << a << ", " << b << std::endl;

std::string A = "foo";
std::string B = "bar";

std::cout << A << ", " << B << std::endl;
swap(A,B);
std::cout << A << ", " << B << std::endl;
```

Example 2: template class to hold a pair of objects

- Templates work with classes as well.
  - For example, we might define a tuple holding two different types (aka `std::pair`)

```
template <typename T1, typename T2>
class pair
{
public:

  pair(const T1 & first, const T2 & second);

  T1 first();
  T2 second();

private:
  const T1 m_first;
  const T2 m_second;
};
```

Example 2: template class to hold a pair of objects

- And implement it like:

```
template <typename T1, typename T2>
pair<T1,T2>::pair(const T1 & first, const T2 & second)
: m_first(first), m_second(second)
{}

template <typename T1, typename T2>
T1 pair<T1,T2>::first()
{
  return m_first;
}

template <typename T1, typename T2>
T2 pair<T1,T2>::second()
{
  return m_second;
}
```

Example 2: template class to hold a pair of objects

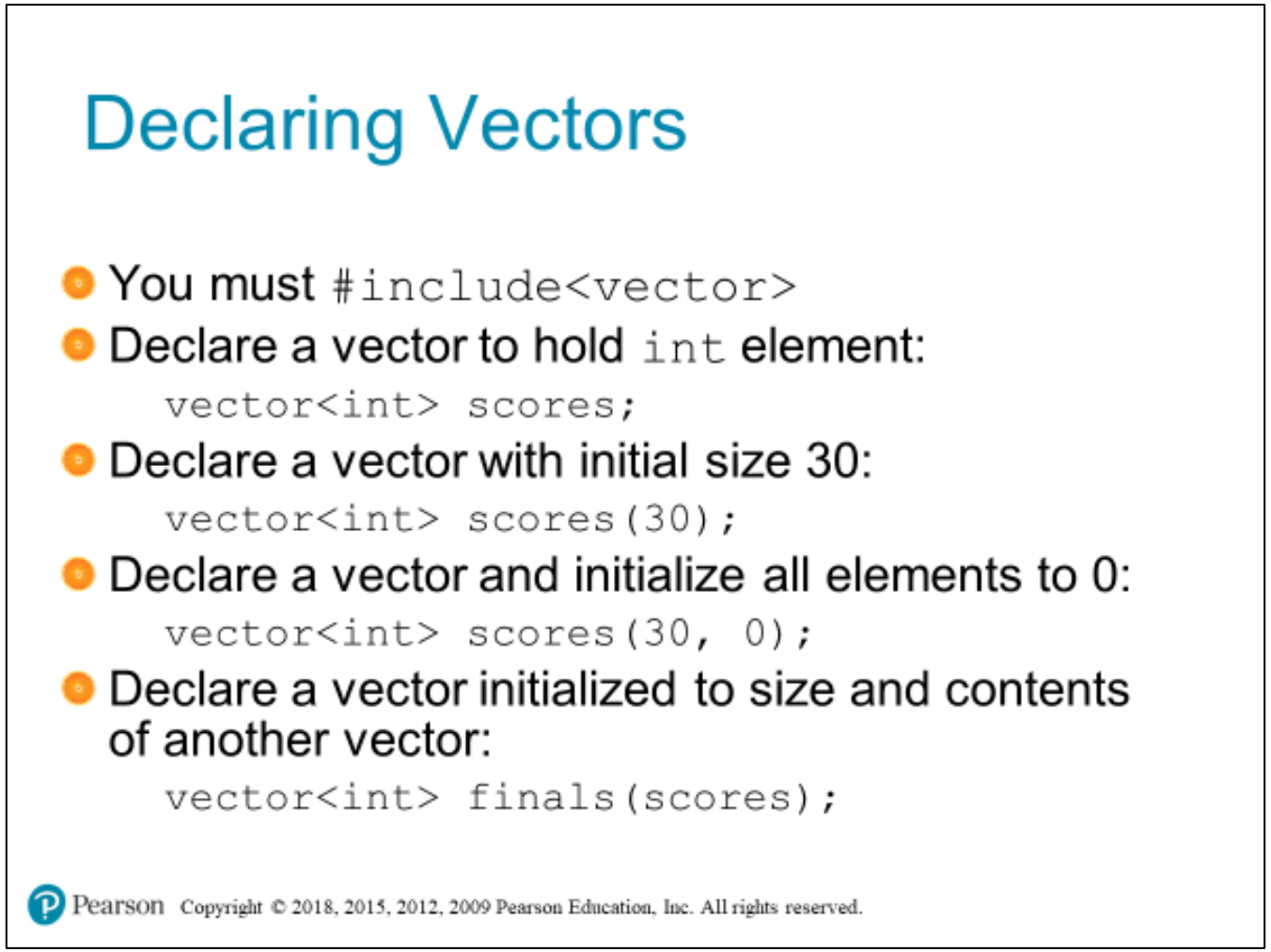
- We might use it like:

```
pair<int,std::string> x(0, std::string("hi"));

std::cout << "First = " << x.first() << std::endl;
std::cout << "Second = " << x.second() << std::endl;
```

STL Vector

- We have seen this before in 301 when we discussed the STL vector:



Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

.txx, .tpp files

- To prevent confusion, another convention is to use a different extension for the template implementation file.
- Examples: .txx, tpp
  - You still include them at the bottom of the header file.

Some tips to make your life (and ours too) easier!

- This is **not** a C++ Programming Class! That was ECE 0301.
  - We assume basic knowledge of C++ syntax and good skills in writing a coherent, readable, and functioning code.
  - Please **debug, debug, and debug** your code, thoroughly before seeking out help from the teaching staff (that includes using as many `printf` as needed to catch the bug).
- Projects and Assignments will **not** be as detailed as it was in ECE 0301.
  - There is some degree of freedom in each assignment. You can make "reasonable" assumptions, as long as they don't contradict with the problem statement.
- When dealing with dynamically allocated arrays, please use the same indexing convention as if it was a statically allocated array.

```
double *arrayPtr = new double[25];
*(arrayptr + 3) = 10;
arrayptr[3] = 10;
```

Assignment/Homework

- **Read:** Carrano pp. 1-24, pp. 31-37 (for next lecture)
- Install Programming Environment
- Introduce Programming Project 1 and In-class Exercise 1.
- **Homework:** Carrano Chapter 1 Exercises 2 and 9
- Chapter 1: Exercise 2:

2. A date consists of a month, day, and year. Consider the class Date of such dates. Suppose that Date represents the month, day, and year as integers. For example, July 4, 1776, is month 7, day 4, and year 1776.

    a. Write specifications for a method within Date that advances any given date by one day. Include a statement of purpose, the preconditions and postconditions, a description of the arguments, and a description of any return value.

    b. Write a C++ implementation of this method. Design and specify any other method that you need. Include comments that will be helpful to someone who will maintain your implementation in the future.

- Chapter1: Exercise 9:

9. Consider the ADT polynomial—in a single variable x—whose operations include the following:

```
degree() // Returns the degree of a polynomial.
coefficient(power) // Returns the coefficient of the  xpower term.
changeCoefficient(newCoefficient, power) // Replaces the coefficient of
                                         // the  xpower term with  newCoefficient.
```

For this problem, consider only polynomials whose exponents are nonnegative integers. For example, $p = 4x^5 + 7x^3 - x^2 + 9$

The following examples demonstrate the ADT operations on this polynomial.

- `p.degree()` is 5 (the highest power of a term with a nonzero coefficient)
- `p.coefficient(3)` is 7 (the coefficient of the $x3$ term)
- `p.coefficient(4)` is 0 (the coefficient of a missing term is implicitly 0)
- `p.changeCoefficient(-3, 7)` changes the polynomial p to $-3x^7 + 4x^5 + 7x^3 - x^2 + 9$

Using these ADT operations, write statements to perform the following tasks:

a. Display the coefficient of the term that has the highest power.

b. Increase the coefficient of the $x3$ term by 8.

c. Compute the sum of two polynomials.