Course Logistics
Welcome to CS 361

• Timings: Monday 4:20pm to 7:00pm

• Location: Diehn 0136

• Instructor: Tamer Nadeem
  Ph.D from Univ. of Maryland, 2006
  Research in Networks, Dist Sys, Mobile Comp.
  Email: nadeem@cs.odu.edu
  Office: ECSB 3204

• Office Hours: Mon 2:30pm-4:00pm, or by appointment
Welcome to CS 361

• Teaching Asst.: Ahmed Salem
  Email: asalem@cs.odu.edu
  Office: ECSB 3106

• Office Hours: Wednesday 10:00am – 1:00pm, or by appointment
Welcome to CS 361

• Prerequisites: CS 250, Problem Solving and Programming, or CS 333, Problem Solving and Programming in C++

I will assume that you are familiar with the basics of C++, including:

• the various C++ statements and control-flow constructs,
• the built-in data types,
• the use of arrays, pointers, pointers to arrays, and linked lists,
• the use and writing of functions, and
• the basic use of structs and classes for implementing abstract data types.
Welcome to CS 361

• Grading:
  • Midterm: 25%
  • Assignments/Programming: 40%
  • Final Exam: 35%

The grading scale is as follows:
(+ and - modifiers will be applied as appropriate)

90-100    A
80-89     B
70-79     C
0-69      F
Welcome to CS 361

• Class Webpage:
  • Please check course website frequently

• Make up classes:
  • Will be occasionally necessary due to travel
Welcome to CS 361

• Text:
  • Required:

• Course Slides (like this one)
Welcome to CS 361

• Academic Integrity / Honor Code:

  • "I pledge to support the honor system of Old Dominion University. I will refrain from any form of academic dishonesty or deception, such as cheating or plagiarism. I am aware that as a member if the academic community, it is my responsibility to turn in all suspected violators of the honor system. I will report to Honor Council hearings if summoned."

  • Please refer to ODU Honor Council’s webpage: http://orgs.odu.edu/hc/
Welcome to CS 361

• Course Policy:
  - Late assignments are not accepted.
  - Attendance
  - Computer Account & Email

Please refer to class webpage for more details.
Course Overview
This Course

• To present the *commonly used data structures*. These form a programmer’s basic data structure “toolkit.” For many problems, some data structure in the toolkit provides a good solution.

• To introduce the idea of tradeoffs and reinforce the concept that there are *costs and benefits associated with every data structure*. This is done by describing, for each data structure, the amount of *space* and *time* required for typical operations.

• To teach *how to measure the effectiveness of a data structure or algorithm*. Only through such measurement can you determine which data structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.
Class Responsibilities

• I will lead lectures

• For every class, read the readings list before the class

• Slides will be posted before of the class time

• Assignments should be on time
Course Structure

• 1 mid term?
  • Tentative date of mid-term:  
    Mon Oct 21 (during class time)

• Assignments / Programming Assignments

• Class ends with the final exam: 
  Mon Dec 09 (3:45pm-6:45pm)
INTRODUCTION
Programming

Program = Algorithm + Data Structure + Implementation

• Algorithm
  • The basic method; it determines the data-items computed.
  • Also, the order in which those data-items are computed (and hence the order of read/write data-access operations).

• Data structures
  • Systematic way of organizing and accessing (read/write) data-items used/computed.

Total Time = Time to access/store data + Time to compute data

Efficient Program = Good algorithm + Good data-structures
Example

• Problem: Compute the average of three numbers.

• Algorithm:  
  (1) aver=(x + y + z)/3  
  (2) aver=(x/3) + (y/3) + (z/3)

  • Method (1) superior to Method (2); two less div-operations.  
  • They access data in the same order: <x, y, z, aver>.

• Data structures:  
  (1) Three variables x, y, z  
  (2) An array nums[0..2]

  • Accessing array-item takes more time than accessing a simple variable (addr(nums[i]) = addr(nums[0]) + i*sizeof(int)).  
  • When there are large number of data-items, naming individual data-items is not practical.
Algorithm Performance

Analytic Method:

• Theoretical analysis of the Algorithm’s time complexity.

Empirical Methods:

• Count the number of times specific operations are performed by executing an instrumented version of the program.

• Measure directly the actual program-execution time in a run.

Example:

Original code: 
if (x < y) small = x;
else small = y;

Instrumented code: 
countComparisons++; //initialized
elsewhere
         
if (x < y) small = x;
else small = y;
Algorithm Testing

• Create a few general examples of input and the corresponding outputs.
  • Select some input-output pairs based on your understanding of the problem and before you design the Algorithm.
  • Select some other input-output pairs after you design the Algorithm, including a few cases that involve special handling of the input or output.

• Use these input-output pairs for testing (but not proving) the correctness of your Algorithm.

• Illustrate the use of data-structures by showing the "state" of the data-structures (lists, trees, etc.) at various stages of the Algorithm.
Data Structures

• A data structure is any data representation and its associated operations.
  • Ex: int, char, float, array, etc.

• A data structure is meant to be an organization or structuring for a collection of data items
  • Ex: vector, struct, class, etc.

How do we organize information so that we can find, update, add, and delete portions of it efficiently?
Data Structures Example Applications

1. How does Google quickly find web pages that contain a search term?

2. What’s the fastest way to broadcast a message to a network of computers?

3. How can a subsequence of DNA be quickly found within the genome?

4. How does your operating system track which memory (disk or RAM) is free?

5. In the game Half-Life, how can the computer determine which parts of the scene are visible?

6. …
Data Structures

• For a data structure, we need to define:
  • how to store a collection of objects in memory,
  • what operations we can perform on that data,
  • the algorithms for those operations, and
  • how time and space efficient those algorithms are.

• Ex. vector in C++:
  • Stores objects sequentially in memory
  • Can access, change, insert or delete objects
  • Algorithms for insert & delete will shift items as needed
  • Space: $O(n)$, Access/change = $O(1)$, Insert/delete = $O(n)$
Abstract Data Type (ADT)

- **ADT:** An *abstract* model that specifies the *structure type* used to store the data and the *operations* that support the data.

- **Abstract?**
  - Encapsulate data and functions that work with this data.
  - Implementation-independent view of the data structure.
  - Interested in what the structure does and not how it does it.
  - Data is not visible to the user and accessed by *operations*.

- **An operation** is a process that can *accept* data values (arguments), *execute* calculations, and *return* a value.
ADT Example

- ADT Dictionary

```cpp
class Dictionary {
    Dictionary();
    void insert(int x, int y);
    void delete(int x);
    ...
}

int main() {
    D = new Dictionary();
    D.insert(3, 10);
    cout << D.find(3);
}
```
ADTs as contracts

An ADT represents a contract between the ADT creator (developer) and the users (application programmers)

• Users of the ADT are expected to alter/examine values of this type only via the operations Provided.

• The creator of the ADT promises to leave the operation specifications unchanged.

• The creator of the ADT is allowed to change the code of the operations at any time, as long as it continues to satisfy the specifications.

• The creator of the ADT is also allowed to change the data structure actually used to implement the type.

• Users can be designing and even implementing the application before the details of the ADT implementation have been worked out.

ADTs designed in this manner are re-usable in which we save time in implementation and debugging.
Abstract Data Type

• An ADT is implemented by supplying
  • A *data structure* for the type name.
  • Coded *algorithms* for the operations.

• Programming language may provide support for communicating ADT to users.
  • In C++, implementation is generally done using a C++ class

In this course, we will study various ADT types implemented using C++ *classes* (e.g., Vectors, Lists, Trees, Sets, Stacks, Queues, etc.)
ADT Implementation in C++
C++ Class

• Each *operation* associated with the ADT is implemented by a *member function* (*method*).

• The *variables* that define the space required by a data item are referred to as *data members*.

• An *object* is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

• Typically, a C++ class has two separated parts: *interface* and *implementation*
  • That *interface* part gives the code for *class declaration*. This code typically goes into a *header file* with the extension *.h*
  • The *implementation* part gives the code for the *function bodies*. This code typically goes in a *source file* with the extension *.c* or *.cc*
Class Declaration

• The declaration of a class includes a **class header** that consists of the reserved word ‘**class**’ followed by the name of the class.

• The rest is the **class body** encloses in braces ‘{’ and terminates with semicolon ‘;’

• The class body includes **private** and **public** sections of **data members** and **member functions** (operations).

```c++
Class className
{
    public:
        // <public member functions prototypes>
        .........
    private:
        // <private data members>
        .........
        // <private member functions prototypes>
        .........
};
```
Class Declaration/Implementation

- **Function Prototype:**
  ```cpp
  returnType functionName(<argument list>);
  ```

- **Function Body:**
  ```cpp
  returnType className::functionName(<argument list>){
      .......
      .......
      .......
  }
  ```

- **Class Instantiation:**
  ```cpp
  main{
      .......
      className classObject1, *classObject2;
      .......
      x = classObject1.functionName(...);
      classObject2 = new className;
      classObject2->functionName(...);
  }
  ```
Example: Class Header Definition

- box.h

```
#include <iostream>

class Box
{
    public:
        // Member functions declaration
        double getVolume(void);
        void setLength( double len );
        void setBreadth( double bre );
        void setHeight( double hei );

    private:
        // Data members
        double length;   // Length of a box
        double breadth;  // Breadth of a box
        double height;   // Height of a box

};
```
Example: Function Bodies

• box.cc

```c++
#include "box.h"

// Member functions definitions
double Box::getVolume(void)
{
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
{
    breadth = bre;
}

void Box::setHeight( double hei )
{
    height = hei;
}
```
Example: Class Instantiation/Usage

• main.cc

```c++
#include "box.h"

// Main function for the program
int main( )
{
    Box box1; // Declare Box1 of type Box
    Box box2; // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    box1.setLength(6.0);
    box1.setBreadth(7.0);
    box1.setHeight(5.0);

    // box 2 specification
    box2.setLength(12.0);
    box2.setBreadth(13.0);
    box2.setHeight(10.0);

    // volume of box 1
    volume = box1.getVolume();
    cout << "Volume of Box1 : " << volume << endl;

    // volume of box 2
    volume = box2.getVolume();
    cout << "Volume of Box2 : " << volume << endl;

    return 0;
}
```
Private and Public Sections

- The public and private sections in a class declaration allow program statements outside the class *different access* to the class members.

```cpp
main{
    ........
    className classObject1, *classObject2;
    ........
    x = classObject1.functionName(...);
    classObject2 = new className;
    classObject2->functionName(...);
}

returnType className::functionName(<argument list>){
    ........
    ........
    ........
}
```
Public Section

• Public members of a class are the *interface* of the object to the program.
  
  • Any statement in a program block that declares an object can access a public member of the object.

```plaintext
main{
  .......
  className  classObject1, *classObject2;
  .......
  x = classObject1.functionName(...);
  classObject2 = new className;
  classObject2->functionName(...);
}
```
Private Section

- The private section typically contains the data values of the object and utility functions that support class implementation.

- Only member functions of the class may access elements in the private section.

```cpp
returnType className::functionName(<argument list>){
    ......
    ......
    ......
}
```
# Concepts of C++ Classes

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class member functions</td>
<td>A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.</td>
</tr>
<tr>
<td>Class access modifiers</td>
<td>A class member can be defined as public, private or protected. By default members would be assumed as private.</td>
</tr>
<tr>
<td>Constructor &amp; destructor</td>
<td>A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.</td>
</tr>
<tr>
<td>C++ copy constructor</td>
<td>The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.</td>
</tr>
<tr>
<td>C++ inline functions</td>
<td>With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.</td>
</tr>
<tr>
<td>The this pointer in C++</td>
<td>Every object has a special pointer this which points to the object itself.</td>
</tr>
<tr>
<td>Pointer to C++ classes</td>
<td>A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.</td>
</tr>
<tr>
<td>Static members of a class</td>
<td>Both data members and function members of a class can be declared as static.</td>
</tr>
</tbody>
</table>
Class Constructors

• A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

• A constructor will have exact same name as the class and it does not have any return type at all, not even void.

• Constructors can be very useful for setting initial values for certain member variables.

```cpp
class Box
{
    public:
        Box(); // This is the constructor
        ....
    private:
        ....
};

// Member functions definitions including constructor
Box::Box(void)
{
    cout << "Box is being created" << endl;
}

....
```
Class Constructors

• main.cc:

```cpp
// Main function for the program
int main() {
    Box box1;

    // set Box length
    box1.setLength(6.0);
    box1.setBreadth(4.0);
    box1.setHeight(2.0);
    cout << "Volume of box : " << box1.getVolume() << endl;

    return 0;
}
```

• Output of program:

Box is being created
Volume of Box : 48
Parameterized Class Constructors

• A class constructor with no argument is called the **default constructor**.

• If you do not define one, the compiler defines an empty one for you.

• A **parameterized** constructor helps you to assign initial value to an object at the time of its creation.

```cpp
class Box
{
  public:
    Box(double len, double brd, double hgt); // parameterized constructor
    ....
  private:
    ....
};
```

```cpp

// Member functions definitions including constructor
Box::Box(double len, double brd, double hgt)
{
  cout << "Box is being created" << endl;
  length = len;
  breadth = brd;
  height = hgt;
}
```
Class Constructors

• main.cc:

```c++
// Main function for the program
int main( )
{
    Box box1 (6.0, 4.0, 2.0);
    cout << "Volume of box : " << box.getVolume() << endl;
    box1.setLength(8.0);
    cout << "Volume of box : " << box.getVolume() << endl;
    return 0;
}
```

• Program Output:

Box is being created
Volume of box : 48
Volume of box : 64
Class Constructors

• In case of parameterized constructor, you can use *Initialization Lists* syntax to initialize the fields:

```cpp
Box::Box(double len, double brd, double hgt) {
    cout << "Box is being created" << endl;
    length = len;
    breadth = brd;
    height = hgt;
}

// Using Initialization List to initialize data members
Box::Box(double len, double brd, double hgt) : length(len), breadth(brd), height(hgt) {
    cout << "Box is being created" << endl;
}
```

Class Destructor

• A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

• A destructor has exact same name as the class prefixed with a tilde ‘~’ and it can neither return a value nor take a parameter.

• Destructor can be very useful for releasing resources before coming out of the program (e.g., closing files, releasing memories, etc.)

```cpp
class Box
{
public:
  Box(); // This is the constructor
  ~Box(); // This is the destructor
private:
  ....
};

Box::~Box(void)
{
  cout << "Box is being deleted" << endl;
}
```

Class Destructor

• main.cc:

```cpp
// Main function for the program
int main( )
{
    Box box1;

    // set Box length
    box1.setLength(6.0);
    box1.setBreadth(4.0);
    box1.setHeight(2.0);
    cout << "Volume of box : " << box1.getVolume() << endl;

    return 0;
}
```

• Program Output:

Box is being created
Volume of box : 48
Box is being deleted
Class Copy Constructor

- The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

- The copy constructor is used to:
  - Initialize one object from another of the same type.
  - Copy an object to pass it as an argument to a function.
  - Copy an object to return it from a function.

- If a copy constructor is not defined in a class, the compiler itself defines one.

- If the class has \textit{pointer variables} with dynamic memory allocations, then it is a \textit{must} to have a copy constructor.

- Most common form of copy constructor

```cpp
classname (const classname &obj) {
  // body of constructor
}
```
//Class Declaration
class Box
{
    public:
        Box(); // Default constructor
        Box(string n); // Constructor
        Box(const Box &obj); // Copy constructor
    void setName(std::string n);
    ~Box(); // Destructor
    ....
    private:
        string *name;
        ....
};

//Function Bodies
Box::Box(std::string n)
{
    cout << "Normal Constructor" << endl;
    name = new std::string;
    *name = n;
}

//Function Bodies
Box::Box(const Box &obj)
{
    cout << "Copy Constructor" << endl;
    name = new std::string;
    *name = *(obj.name);
}

//Function Bodies
Box::~Box()
{
    cout << "Destructor – Freeing Memory" << endl;
    if (name) delete name;
}

//Function Bodies
Box::setName(std::string n)
{
    cout << "Setting Name…" << endl;
    *name = n;
}
Class Copy Constructor

• main.cc:

```cpp
// Main function for the program
Void display(Box obj)
{
    cout << ”Name of box : " << obj.getName() <<endl;
}

int main() {
    Box box1(“Name1”);
    Box box2 = box1;

    display(box1);
    display(box2);
    box2.setName(“Name2”);
    display(box1);
    display(box2);

    return 0;
}
```

• Program Output:

```
Normal Constructor
Copy Constructor
Copy Constructor
Name of box : Name1
Destructor – Freeing Memory
Copy Constructor
Name of box : Name1
Destructor – Freeing Memory
Setting Name…
Copy Constructor
Name of box : Name1
Destructor – Freeing Memory
Copy Constructor
Name of box : Name2
Destructor – Freeing Memory
Destructor – Freeing Memory
```

.. .

// Main function for the program
Void display(Box obj)
{
    cout << ”Name of box : " << obj.getName() <<endl;
}

int main() {
    Box box1(“Name1”);
    Box box2 = box1;

    display(box1);
    display(box2);
    box2.setName(“Name2”);
    display(box1);
    display(box2);

    return 0;
}
Class Inline Functions

• Many of the member functions in this example are simple enough that we might consider an alternate approach, declaring them as *inline functions*.

• A function definition in a class definition is an inline function definition (no need to use *inline* tag).

• Inline functions are suggestions to the compiler that calls to these functions can be compiled in away that maximizes speed at the expense of code size.

• Typically, if a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
Inline Functions

• Example: Box Class

```cpp
class Box {
    public:
    // Member functions declaration
    double getVolume(void) {return length * breadth * height;}
    void setLength( double len ) {length = len;}
    void setBreadth( double bre ) {
        breadth = bre;
    }
    void setHeight( double hei ) {height = hei;}

    private:
    // Data members
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};
```
Class Operators Overloading

- You can redefine or overload most of the built-in operators available in C++.

- Overloaded operators are functions with special names the keyword `operator` followed by the `symbol` for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

- Typically, most overloaded operators be defined as **ordinary non-member functions** (typically as `friend` functions) or as **class member functions**.

- Ex: declares the addition operator that can be used to add two Box objects and returns final Box object.
  - **Class member function:**
    ```cpp
    Box operator+(const Box&);
    ```
  - **Ordinary non-member function:**
    ```cpp
    Box operator+(const Box&, const Box& b);
    ```


Class Operator Overloading

• Example: Box Class

```cpp
// Member functions declaration
double getVolume(void) {return length * breadth * height;}

// Overload + operator to add two Box objects.
Box operator+(const Box& b) {
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}
```

```cpp
// Main function for the program
int main() {
    Box Box1; // Declare Box1 of type Box
    Box Box2; // Declare Box2 of type Box
    Box Box3; // Declare Box3 of type Box

    // Add two object as follows:
    Box3 = Box1 + Box2;
```

```cpp
    return 0
}
```
Class Operator Overloading

Following is the list of operators which can be overloaded:

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
<td>!</td>
<td>~</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>=</td>
<td>!=</td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
</tr>
<tr>
<td>/=</td>
<td>*=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>[]</td>
<td>()</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;*</td>
<td>new</td>
<td>new []</td>
<td>delete</td>
<td>delete []</td>
</tr>
</tbody>
</table>

Following is the list of operators which can not be overloaded:

:: . * ?:
Class Operator: Assignment Operator

- One of the most common operator to overload is the assignment operator ‘=‘.

- Assignment operator is used to assign values of a class object to another.

- If a class does not provide an explicit assignment operator, the compiler will generates one. The compiler-generated version will simply assign each data member in turn.

- If the class has `pointer variables` with dynamic memory allocations, then it is a `must` to have a copy constructor.

  - `Class member function:`

    ```cpp
    Box& operator=(const Box&);```

    ```cpp
    Box& operator=(const Box&);```
Class Operator: I/O Operators

• Very common set of operators that programmers often write for their own code are the I/O operators ‘<<’ and ‘>>’, particularly the output operator ‘<<’

• It is a goof practice always provide an output operator for every class you write, even if you don’t expect to use it in your final application. This output operators could be used during debugging your program.

• Only defined as ordinary non-member function as a friend function to the class

• Ex: declares the output operator that can be used to print the Box information.

```cpp
friend ostream&  operator<<(ostream& out, const Box& b);
```
Class Operator: Comparison Operators

- After assignments and I/O, the most commonly programmed operators ‘would be’ the *relational operators*, especially ‘==’ and ‘<’.
- The compiler never generates these implicitly, so if we want them, we have to supply them.
- If the class has *pointer variables* with dynamic memory allocations, then it is a *must* to have relation operators.
- Ex: declares the equality operator that can be used to compare two Boxes.
  - *Class member function:*
    ```cpp
    bool operator==( const Address&) const ;
    ```
Class Const Correctness

- In C++, we use the keyword ‘const’ in member-functions of a class to:
  1. Indicate what formal parameters a function will look at, but promises not to change
  2. Indicate which member functions don’t change the object they are applied to

```cpp
....
class Box
{
  public:
    // Member functions declaration
    double getVolume(void) const {return length * breadth * height;}
    void setLength(double len) {length = len;}
    ....
    // Overload operators.
    Box operator+(const Box& b);
    bool operator==(const Box& b) const;
  private:
    ....
};
```
Questions?
Assignment #1: Warming UP

• Due Wed Sep 4th, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #1:
    
<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>43</td>
</tr>
<tr>
<td>16</td>
<td>43</td>
</tr>
<tr>
<td>17</td>
<td>44</td>
</tr>
<tr>
<td>24</td>
<td>46</td>
</tr>
<tr>
<td>28</td>
<td>47</td>
</tr>
</tbody>
</table>

• Programming Assignment
  Ford & Topp, Chapter #1: Q #47, P #52.
Assignment #1: Warming UP

• Due Wed Sep 4th, 11:59pm

• Submission Format:

  • Written Assignment
    • Create single PDF file with name: cs361_assignment_1_<firstName>_lastName>
    • Have a cover page with your name and your email
    • Send the file as an attachment to TA
    • Make the subject line of the email: cs361_assignment_1

  • Programming Assignment
    • Make sure your program compiles and executes using g++ on Dept’s Linux machines.
    • Create a “Readme.txt” file that list how to compile and execute your program. Include your name and your email.
    • Zip all files (.h, .c, Makefile, etc.) and name it: cs361_program_1_<firstName>_lastName>
    • Send the zip file as an attachment to TA
    • Make the subject line of the email: cs361_program_1