- 3 Parts of an ADT
  - Data- some memory
  - Operations on the data- set of functions that work on the data
  - Rules of Usage-rules that must be followed
- Failure vs faults
  - Failure: incorrect output for given input
  - Fault: incorrect piece of code
- Debugging basics
  - Assert: statement that is true at a specific point in the program
  - Pre-condition (REQUIRES): statement that must be true before a function is called
  - Post-condition (ENSURES): statement that is true after function is executed
  - Invariants: statements that are always true
    - LOOP INV: true within the scope of the loop
    - CLASS INV: true for object of a class
    - GLOBAL INV: true globally
- Const methods
  - Do NOT modify the object they are being called on
- Pointers
  - Address of operator: &
  - Dereference operator: *
  - Cannot return local variables by reference
  - Cannot return pointers to local variables
  - Arrays are pointers
  - *I don't think he will ask about the dynamic stuff like new and delete but go over it just in case*
- Set ADT

```
#include <iostream>
#include <initializer_list>

const int Set_domain=6400;
//Class INV: all elements must be between 0 and domain-1
class Set {
public:
        Set();
        Set(int);
        Set(int, int);
        Set(std::initializer_list<int>);

        int card() const;
        Set operator*(const Set&) const; //intersection
        Set operator+(const Set&) const; //Union
        Set operator-(const Set&) const;  //difference
        bool operator[](const int&) const;
```

```
        bool operator==(const Set&) const;
        bool operator<=(const Set&) const; //subset

private:
        bool element[Set_domain]
};

std::ostream& operator<<(std::ostream&, const Set&)

Set operator*(int, const Set&);
Set operator+(int, const Set&);
Set operator-(int, const Set&);
bool operator==(int, const Set&);
bool operator !=(const Set&, const Set&);
bool operator<=(int, const Set&);
bool operator<(const Set&, const Set&);
bool operator>=(const Set&, const Set&);
bool operator>(const Set&, const Set&);
#endif

#include "set.hpp"

bool isValid(int x) {
        if ((x >= 0) && (x < Set-domain)) return true;
        else return false;
}

Set::Set() {
        for (int i = 0; i < Set_domain; ++i) {
                element[i] = false;
        }
}

//REQUIRES: 0<=a<domain
Set::Set(int a) :Set() { //delegation
        if (isValid(a)) element[a] = true;
}
//REQUIRES: 0<=a,b<domain
Set::Set(int a, int b) :Set() {
        if (isValid(a)) element[a] = true;
        if (isValid(b)) element[b] = true;
}

Set::Set(std::initializer_list<int> lst) : Set() {
```

```
	for (int i : lst) {
		if (isValid(i)) element[i] = true;
	}
}

int Set::card() const {
	int result = 0;
	for (int i = 0; i < domain; ++i) {
		if (element[i]) ++result;
	}
	return result;
}
//REQUIRES: 0<=i<domain
bool Set::operator[](int i) const {
	if (isValid(i))
		return element[i];
	else
		return false;
}
//intersection
Set Set::operator*(const Set& rhs) const {
	Set result;
	for (int i = 0; i < Set_domain; ++i) {
		result.element[i] = element[i] && rhs.element[i];
	}
	return result;
}
//union
Set Set::operator+(const Set& rhs) const {
	Set result;
	for (int i = 0; i < Set_domain; ++i) {
		result.element[i] = element[i] || rhs.element[i];
	}
	return result;
}

Set Set::operator- (const Set & rhs) const {
	Set result;
	for (int i = 0; -< Set_domain; ++i) {
		result.element[i] = element[i] && !rhs.element[i];
	}
	return result;
}
```

```
Set operator+(int lhs, const Set& rhs) {
	return Set(lhs) + rhs;
}

Set operator*(int lhs, const Set& rhs) {
	return Set(lhs) * rhs;
}

Set operator-(int lhs, const Set& rhs) {
	return Set(lhs) - rhs;
}

bool Set::operator==(const Set& rhs) const {
	for (int i = 0; i < Set_domain; ++i) {
		if (element[i] != rhs.element[i])
			return false;
	}
	return true;
}

bool operator==(int lhs, const Set& rhs) {
	return Set(lhs) == rhs;
}

bool operator!=(const Set& lhs, const Set& rhs) {
	return !(lhs == rhs);
}

//subset
bool Set::operator<=(const Set& rhs) const {
	for (int i = 0; i < Set_domain; ++i) {
		if (element[i] && !rhs.element[i])
			return false;
	}
	return true;
}

bool operator<(const Set& lhs, const Set& rhs) {
	return !(rhs <= lhs);
}

bool operator>=(const Set& lhs, const Set& rhs) {
	return rhs <= lhs;
}
```

```
bool operator>(const Set& lhs, const Set& rhs) {
	return !(lhs <= rhs);
}

std::ostream& operator<<(std::ostream& out, const Set& rhs) {
	out << "{";
	bool printComma = false;
	for (int i = 0l i < Set_domain; ++i) {
		if (rhs[i]) {
			if (printComma) out << ", ";
			out << i;
			printComma = true;
		}
	}
	out << "}";
	return out;
}
```

- Point ADT

```
#include "point.hpp"

int main() {

Point a;

Point b(10,20);

a.init();

b.init(20,40);

cout << a.getx() << endl;

a.setx(10);

a.sety(20);

a.print(cout);

std::cout << a; //operator<<(std:cout, a)

// operator<<(operator<<(std::cout, a), std::endl);
```

```
ofstream out("temp.txt");
a.print(out);
return 0;
}
```

```
#ifndef CS2_POINT_HPP_
#define CS2_POINT_HPP_
#include <iostream>
#include <cmath>
#include <fstream>
class Point {
public:
Point();
Point(double, double);
void init();
void init(double, double);
double getx();
double gety();
void setx(double nx);
void sety(double);
void print(std::ostream&); //void operator<<(std::ostream&);
Point operator+(Point rhs);
Point operator-(Point rhs);
```

```
Point add(Point rhs);
Point sub(Point rhs);
double dist(Point);
private:
double x, y;
};
std::ostream& operator<<(std::ostream&, Point);
#endif
```

```
#include "point.hpp"
Point::Point() {
x = 0;
y = 0;
}
void Point::init() {
 x = 0;
 y = 0;
}
void Point::init(double nx, double ny) {
x = nx;
y = ny;
}
double Point::getx() {
```

```
return x;
 }
double Point::gety() {
return y;
}
void Point::setx(double nx) {
x = nx;
}
void Point::sety(double ny) {
y = ny;
}
Point Point::operator+(Point rhs) {
Point result;
result.x = x + rhs.x;
result.y = y + rhs.y;
return result;
}
Point Point::add(Point rhs) {
Point result;
result.x = x + rhs.x;
result.y = y + rhs.y;
return result;
}
```

```
Point Point::operator-(Point rhs) {
Point result;
result.x = x - rhs.x;
result.y = y - rhs.y;
return result;
}
Point Point::sub(Point rhs) {
Point result;
result.x = x - rhs.x;
result.y = y - rhs.y;
return result;
}
double Point::dist(Point rhs) {
return (x - rhs.x)(x - rhs.x) - (y - rhs.y)(y - rhs.y);
}
void Point::print(std::ostream& out) {
out << "(" << x << ", " << y << ")";
}
std::ostream& operator<<(std::ostream& out, Point rhs) {
out << "(" << rhs.getx() << ", " << rhs.gety() << ")";
return out;
}
```

- String ADT

```
#ifndef STRING_H_
#define STRING_H_
#include <iostream>
#include <initializer_list>
#include <cassert>

const int STRING_SIZE = 256; //string capacity+1 (for null terminator)
//CLASS INV: 0<=length()<=capactity()&& capacity()==STRING_SIZE-1&&str[length()]==0&&
can only access str[0,...,length()-1]
class String {
public:
	String();
	String(const char[]);
	String(char);

	int length() const;
	int capacity() const { return STRING_SIZE - 1; }
	char operator[](int) const;
	char& operator[](int);

	String operator+(const String& ) const;
	String& operator+=(const String&);
	bool operator==(const String&) const;
	bool operator<(const String&) const;
	String substr(int start, int end) const;
	int findchar(int start, char key) const;
	int findStr(int start, const String& key) const;

	friend std::ostream& operator<<(std::ostream&, const String&);

private:
	char str[STRING_SIZE];
};

//free function
std::istream& operator>>(std::istream&, String&);

String operator+(String, const String&);

bool operator==(const char[], const String&);
bool operator==(char, const String&);
bool operator<(const char[], const String&);
bool operator<(char, const String&);
bool operator !=(const String&, const String&);
```

```
bool operator >(const String&, const String&);
bool operator <=(const String&, const String&);
bool operator >=(const String&, const String&);

#endif

#include "string.h""

String::String() {
	str[0] = 0;
}

String::String(char ch) {
	str[0] = ch;
	str[1] = 0;
}

String::String(const char s[]) {
	int i = 0;
	while (s[i] != 0) {
		if (i >= STRING_SIZE - 1) break;
		str[i] = s[i];
		++i;
	}
	str[i] = 0;
}

int String::length() const {
	int len = 0;
	while (str[len] != 0) ++len;
	return len;
}

//REQUIRES: 0<=i<length
char String:: operator[](int i) const {
	assert((i >= 0) && (i < length()));
	return str[i];
}

//REQUIRES: 0<=i<length
char& String::operator[](int i) {
	assert((i >= 0) && (i < length()));
	return str[i];
}
```

```
String String::operator+(const String& rhs) const { //typically would not implement this
	String result(str);
	int offset = length(); //implicitly calls on left hand side since there is no . operator
	int i = 0;
	while (rhs.str[i] != 0) {
		if (offset + i >= STRING_SIZE - 1) break;
		result.str[offset + i] = rhs.str[i];
		++i;
	}
	result.str[offset + i] = 0;
	return result;
}

String& String::operator+=(const String& rhs) { //not a const method
	int offset = length();
	int rhslen = rhs.length();
	int i = 0;
	while (i < rhslen) {
		if (offset + i >= STRING_SIZE - 1) break;
		str[offset + i] = rhs.str[i];
		++i;
	}
	str[offset + i] = 0;
	return *this;
}

String operator+(String lhs, const String& rhs) {
	return lhs += rhs;
}

bool String::operator==(const String& rhs) const {
	int i = 0;
	while (str[i] != 0 && str[i] == rhs.str[i]) ++i;
	return str[i] == rhs.str[i];
}

bool String::operator<(const String& rhs) const {
	int i = 0;
	while ((str[i] !=0)&& (rhs.str[i] !=0) && (str[i]== rhs.str[i])) ++i;  //'A' < 'B' ; "ABC"<"ABCD"
will not compare lengths in this case but still good to know
	return str[i]<rhs.str[i];
}
```

```
bool operator==(char lhs, const String& rhs) { return String(lhs) == rhs; }
bool operator==(const char lhs[], const String& rhs) { return String(lhs) == rhs; }

bool operator<(char lhs, const String& rhs) { return String(lhs) < rhs; }
bool operator<(const char lhs[], const String& rhs) { return String(lhs) < rhs; }

bool operator!=(const String& lhs, const String& rhs) { return !(lhs == rhs); }

bool operator>(const String& lhs, const String& rhs) { return rhs < lhs; }

bool operator<=(const String& lhs, const String& rhs) { return !(rhs < lhs); }

bool operator>=(const String& lhs, const String& rhs) { return !(rhs > lhs); }

//REQUIRES: 0<=start<=end<length()
//ENSURES: RetVal==str[start,...,end]
String String::substr(int start, int end) const {
	String result;
	if (start < 0) start = 0;
	if (end >= length()) end = length() - 1;
	if (start > end) return String();
	int i;
	for (int i = start; i <= end; ++i) {
		result.str[i-start] = str[i];
	}
	result.str[i-start] = 0;
	return result;
}

//REQUIRES: 0<= start<length()
//ENSURES: RetVal==i where str[i]==key&& i>=start ||RetVal==-1 where
key!=str[start,...,length()-1]
int String::findchar(int start, char key) const {
	if (start < 0) start = 0;
	if (start >= length()) return -1;
	int i = start;
	while (str[i] != 0) {
		if (str[i] == key) return i;
		++i;
	}
	return -1;
}

//example of friend function
```

```
std::ostream& operator<<(std::ostream& out, const String& rhs) {
        out << rhs.str;
        return out;
}

std::istream& operator>>(std::istream& in, String& rhs) {
        char buffer[STRING_SIZE];
        in >>buffer; //Skip leading ws, read until white space
        rhs = String(buffer);
        return in;
}
```

- Bigint ADT (Project 1) I will not share the code to this but I strongly recommend reviewing this

---

Practice Questions

1. Given the class definition below, write a member function that checks if two strings (as defined below) are equal. Give REQUIRES and ENSURES conditions for the operator== you write.

```
class String {
public:
string() { s[0] = 0; };
bool operator==(const String&) const;
private:
char s[256]; //null terminated character array
};
```

```
//REQUIRES: String needs to be a valid string, state of the object before the function is called
//ENSURES: returns true if the objects are equal
bool String::operator==(const String& rhs) const {
        int i = 0;
        while (str[i] != 0 && rhs,str[i] !=0 ) ++i;
        return str[i] == rhs.str[i];
}
```

2. What are the three components of an abstract data type (ADT)? *ON EXAM*

1. Data
2. Operations on the data
3. Rules of Usage

3. Overload operator>> for the string class defined in problem 1. You can assume it is a friend function. Read in a string from a stream until a semicolon (;) is read. Blanks MUST be included into the string, but the end of line character or semicolon should NOT. You do not need to check for end of file.

ANSWER

```
std::istream& operator>>(std::istream& in, bigint& rhs)//should be string not bigint
{
   char ch;
   // max_size will be like the length defined in the bullshit
   char tmp[MAX_SIZE];

   for (int i = 0; i < MAX_SIZE; ++i)
   {
      tmp[i] = 0;
   }

   // read character
   in >> ch;

   int count = 0;
   while (ch != ';' && count < MAX_SIZE)
   {

      tmp[count] = ch;
      ++count;
         In >> ch;
   }

   // not sure if we need this or not
   rhs = bigint(tmp);  //String not bigint

   return in;
}
```

```
std::istream& operator>>(std::istream& in, String& rhs) {
        char newArray[STRING_SIZE];
        char ch;
        int index=0;
        while(in>>ch){
                if(ch=';') break;
                newArray[index++]=ch;
```

```
        }
        rhs=String(newArray);
        return in;
}
```

4. Overload operator<< for the bigint class in project 1. You can assume it is a friend function. Output such that there are a maximum of 60 digits per line. Assume there is a constant value called BIGINT_CAPACITY.

ANSWER

```
std::ostream& <<operator(std::ostream& out, const bigint& big_arr)

{
   int new_size = BIGINT_CAPACITY;

  do
  {
     --new_size;
  } while(new_size > 0 && big_arr[new_size] == 0)


Int count = 0;
  while (new_size >= 0)
  {
     if (count % 80 == 0)
     {
        out << big_arr[new_size];
        out << "\n";
     }
     else
     {
        out << big_arr[new_size];
     }
     --new_size;
     ++ count;
  }

Return out;
}
```

5. Given the class definition below, along with the class invariant, implement 4 methods: the default constructor, a constructor that converts a char[] into a string, the length function which returns the number of characters in the string and operator+ it returns the concatenation of two strings. You can only use these provided methods and *cannot* use any built in functions or libraries. Implement the two string::operator[] methods, a const and non-const version. B) Why do we need to implement both versions?

```
const int CAP = 256;

//CLASS INV: s[length()] == 0, A null terminating char array

class String {

public:

String   ();                    //Need to implement (5pts)

 String    (const char[]);          //Need to implement (10pts)

  int    length    ()  const;   //Need to implement (5pts)

  String operator+ (const String&) const;   //Need to implement (10pts)

   private:

  char   s[CAP];

};
```

```
//Default Constructor
String:: String() {
        str[0]=0;
}

//Constructor that converts a char[] into a string
String:: String(const char s[]) {
        int i=0;
        While (str[i] !=0) {
                if (i>= CAP-1) break;
                str[i]= s[i];
                ++i;
        }
str[i]=0;
}

//Length
int String:: length() const{
        int len=0;
        while (str[len] !=0) ++len;
```

```
        return len;
}

//Overload + which concatenates 2 strings I'm not sure if this is correct
String String::operator+(const String& rhs) const{
        String result(str);
        int offset=length();
        int i=0;
        while(rhs.str[i] !=0) {
                if(offset+i >=CAP-1) break;
                result.str[offset+i]= rhs.str[i];
                ++i;
        }
        result.str[offset+i]=0;
        return result;
}

//The 2 [] operator methods
//We need to implement both so it can be used on const and non-const objects
char String::operator[](int i) const{
        assert((i>=0)&&(i<length()));
        return str[i];
}

char& String::operator[](int i) {
        assert((i>=0)&&(i<length()));
        return str[i];
}
```