CISS350: Data Structures and Advanced Algorithms
Assignment 1

Name: _____

Q1. [Review of CISS240]

Here's a very famous unsolved problem in math and computer science. Consider the function f defined as follows:

1. If n is even, then
$$f(n) = n/2$$
2. If n is odd, then
$$f(n) = 3n + 1$$

For instance
$$f(7) = 22 \text{ and } f(22) = 11$$

There is something very special about this function f. Note that if you start with $n = 7$ and continually apply f you get this:

$$\begin{aligned}
f(7) &= 7 \times 3 + 1 = 22 \\
f(22) &= 22/2 = 11 \\
f(11) &= 11 \times 3 + 1 = 34 \\
f(34) &= 34/2 = 17 \\
f(17) &= 3 \times 17 + 1 = 52 \\
f(52) &= 52/2 = 26 \\
f(26) &= 26/2 = 13 \\
f(13) &= 13 \times 3 + 1 = 40 \\
f(40) &= 40/2 = 20 \\
f(20) &= 20/2 = 10 \\
f(10) &= 10/2 = 5 \\
f(5) &= 3 \times 5 + 1 = 16 \\
f(16) &= 16/2 = 8 \\
f(8) &= 8/2 = 4 \\
f(4) &= 4/2 = 2 \\
f(2) &= 2/2 = 1
\end{aligned}$$

and you reach the value of 1. A very famous conjecture says that no matter what positive integer $n > 0$ you start with, you will always reach 1 if you apply enough times of f. In the above case of $n = 7$, you need to apply f sixteen times to reach 1, i.e.

$$f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(7))))))))))))))))$$

Write a program that accepts a positive integer value for n from the user and prints n, the values obtained from continually applying function f, and the number of times function f was applied. Your code should include a (C++) function

```
int f(int n);
```

which computes just like the above (mathematical) function f.

Test 1

```
1
1 0
```

[Note: Underlined text refers to input. In the above test case, the user enters 1 and press the enter key.]

Test 2

```
7
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 16
```

Q2. [Review of CISS245]

The following is a generalization of Q1. Given positive integers a, b, d, we define function f as follows:

1. If n is divisible by d, then
$$f(n) = n/d$$
2. Otherwise
$$f(n) = an + b$$

(This is a generalization of the function f in Q1: if $a = 3, b = 1, d = 2$, you get the function in Q1.)

Write a program that accepts positive integer values for a, b, d, n, and N (I'll explain N in a bit) and does the following.

First, if you can reach 1 after at most N applications of function f, you print a, b, d, n, N, the string `pass`, the number of tries, and the list of numbers obtained. For instance:

```
3 1 2 7 1000
3 1 2 7 1000 pass 16 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Second, if you cannot reach 1 after N tries, you print the same information as above except that you print `fail` instead of `pass`. For instance:

```
3 1 2 7 10
3 1 2 7 10 fail 10 7 22 11 34 17 52 26 13 40 20 10
```

Note that in this case after applying this particular f 10 times to 7, you have 11 numbers (including 7) and the last is not 1 (it's 10).

In this case you must write a (C++) function f that accepts a, b, d in such a way that the f function call in Q1 still works, i.e. your function prototype must be

```
int f(int n, int a=3, int b=1, int d=2);
```

There is another function `fs` that accepts a pointer to int array,

```
int fs(int * & p, int n, int a=3, int b=1, int d=2, int N=1);
```

allocates an integer array of size N to p. Note that the value of n is not stored in this array. The return value is the number of integers placed in the array. For instance when $a = 3, b = 1, d = 2, n = 7, N = 1000$ (see above test case), there are 17 values

(including $n = 7$). The values stored in the array `p` points to are

```
22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

(i.e. 7 is not included) and hence the return value in this case is 16.

Make sure you deallocate memory `p` points to before program halts.

Test 1

```
3 1 2 7 1000
3 1 2 7 1000 pass 16 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Test 2

```
3 1 2 7 10
3 1 2 7 10 fail 10 7 22 11 34 17 52 26 13 40 20 10
```

Test 3

```
7 5 3 7 10
7 5 3 7 10 fail 10 7 54 18 6 2 19 138 46 327 109 768
```

In the case of Test 3 the function f is defined as

1. If n is divisible by 3, then $f(n) = n/3$
2. If n is not divisible by 3, then $f(n) = 7n + 5$

Therefore we have

$$\begin{aligned} f(7) &= 7 \times 7 + 5 = 54 \\ f(54) &= 54/3 = 18 \\ f(18) &= 18/3 = 6 \\ f(6) &= 6/3 = 2 \\ f(2) &= 2 \times 7 + 5 = 19 \\ f(19) &= 3 \times 19 + 5 = 138 \\ f(138) &= 138/3 = 46 \\ f(46) &= 7 \times 46 + 5 = 327 \\ f(327) &= 327/3 = 109 \\ f(109) &= 7 \times 109 + 5 = 768 \end{aligned}$$

Q3. [Review of CISS245]

This is a continuation of Q2. This is essentially the same as Q2 except that instead of testing a single n, you test a range of values of n. In this case you prompt the user for $a, b, d, n0, n1, N$ and essentially test the given function (which depends on a, b, c) for $n = n0, n0 + 1, n0 + 2, ..., n1$ (inclusive).

Test 1

```
3 1 2 1 9 1000
3 1 2 1 1000 pass 0 1
3 1 2 2 1000 pass 1 2 1
3 1 2 3 1000 pass 7 3 10 5 16 8 4 2 1
3 1 2 4 1000 pass 2 4 2 1
3 1 2 5 1000 pass 5 5 16 8 4 2 1
3 1 2 6 1000 pass 8 6 3 10 5 16 8 4 2 1
3 1 2 7 1000 pass 16 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
3 1 2 8 1000 pass 3 8 4 2 1
3 1 2 9 1000 pass 19 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Test 2

```
3 1 2 1 9 5
3 1 2 1 5 pass 0 1
3 1 2 2 5 pass 1 2 1
3 1 2 3 5 fail 5 3 10 5 16 8 4
3 1 2 4 5 pass 2 4 2 1
3 1 2 5 5 pass 5 5 16 8 4 2 1
3 1 2 6 5 fail 5 6 3 10 5 16 8
3 1 2 7 5 fail 5 7 11 22 34 17 52
3 1 2 8 5 pass 3 8 4 2 1
3 1 2 9 5 fail 5 9 28 14 7 22 11
```

Q4. [Review of CISS245]

This is a very minor modification to Q3. The program now accepts a boolean value (besides the values described in Q3) whether or not to print the number of tries and the list of numbers obtained.

Test 1

```
3 1 2 1 9 1000 1
3 1 2 1 1000 pass 0 1
3 1 2 2 1000 pass 1 2 1
3 1 2 3 1000 pass 7 3 10 5 16 8 4 2 1
3 1 2 4 1000 pass 2 4 2 1
3 1 2 5 1000 pass 5 5 16 8 4 2 1
3 1 2 6 1000 pass 8 6 3 10 5 16 8 4 2 1
3 1 2 7 1000 pass 16 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
3 1 2 8 1000 pass 3 8 4 2 1
3 1 2 9 1000 pass 19 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

In the case of Test 1, the last input value is 1 which indicates a request to print the values after the "pass/fail" column.

Test 2

```
3 1 2 1 9 1000 0
3 1 2 1 1000 pass
3 1 2 2 1000 pass
3 1 2 3 1000 pass
3 1 2 4 1000 pass
3 1 2 5 1000 pass
3 1 2 6 1000 pass
3 1 2 7 1000 pass
3 1 2 8 1000 pass
3 1 2 9 1000 pass
```

In the case of Test 2, the last input value is 0 which indicates a request not to print the values after the "pass/fail" column.

Test 3

```
3 1 2 1 9 5 1
3 1 2 1 5 pass 0 1
3 1 2 2 5 pass 1 2 1
3 1 2 3 5 fail 5 3 10 5 16 8 4
3 1 2 4 5 pass 2 4 2 1
3 1 2 5 5 pass 5 5 16 8 4 2 1
3 1 2 6 5 fail 5 6 3 10 5 16 8
3 1 2 7 5 fail 5 7 22 11 34 17 52
3 1 2 8 5 pass 3 8 4 2 1
3 1 2 9 5 fail 5 9 28 14 7 22 11
```

Test 4

```
3 1 2 1 9 5 0
3 1 2 1 5 pass
3 1 2 2 5 pass
3 1 2 3 5 fail
3 1 2 4 5 pass
3 1 2 5 5 pass
3 1 2 6 5 fail
3 1 2 7 5 fail
3 1 2 8 5 pass
3 1 2 9 5 fail
```

Q5. [Review of classes of CISS245]

This is a review of writing a class (without pointers).

Write a `Fraction` class that should do the obvious. No skeleton code is given. The header file is named `Fraction.h` and the implementation file is named `Fraction.cpp`. You must design and implement the class yourself. You should of course test your code, i.e., you should have a `main.cpp` that tests the features of your fraction library. Try not to refer to your previous work if you have already implemented such a class – otherwise it defeats the purpose of review. You can however refer to my notes on object-oriented concepts/syntax of C++. Only refer to your previous code briefly for the `Fraction` if you are totally lost. You should be able to complete this library in less than 1 hour.

The following illustrates features of this class.

The basic constructor should work as expected:

```
Fraction f(2, 3);
Fraction g(42);
Fraction h;                                 // default constructor

std::cout << f << '\n';                     // prints "2/3" (without the double quotes)
std::cout << g << '\n';                     // prints "42"
std::cout << h << '\n';                     // prints "0"

std::cout << Fraction(2, 4) << '\n';        // prints "1/2"
std::cout << Fraction(-2, 4) << '\n';       // prints "-1/2"
std::cout << Fraction(2, -4) << '\n';       // prints "-1/2"
std::cout << Fraction(-2, -4) << '\n';      // prints "1/2"

std::cout << Fraction(2, 2) << '\n';        // prints "1"
std::cout << Fraction(2, -2) << '\n';       // prints "-1"
std::cout << Fraction(-2, 2) << '\n';       // prints "-1"
std::cout << Fraction(-2, -2) << '\n';      // prints "1"

std::cout << Fraction(100, 20) << '\n';     // prints "5"
std::cout << Fraction(100, -20) << '\n';    // prints "-5"
std::cout << Fraction(-100, 20) << '\n';    // prints "-5"
std::cout << Fraction(-100, -20) << '\n'; // prints "5"

std::cout << Fraction(100) << '\n';         // prints "100"
std::cout << Fraction(-100, 1) << '\n';     // prints "-100"
std::cout << Fraction(100, -1) << '\n';     // prints "-100"
std::cout << Fraction(-100, -1) << '\n';    // prints "100"

std::cout << Fraction() << '\n';            // prints "0"
std::cout << Fraction(0, 100) << '\n';      // prints "0"
```

```
std::cout << Fraction(0, -100) << '\n';   // prints "0"


// Testing division by zero exception catching
try
{
    std::cout << Fraction(0, 0) << '\n';
}
catch (ZeroDivisionError & e)
{
    std::cout << "ZeroDivisionError exception caught\n";
}

try
{
    std::cout << Fraction(-10, 0) << '\n';
}
catch (ZeroDivisionError & e)
{
    std::cout << "ZeroDivisionError exception caught\n";
}
```

Note that the numerator and denominator are reduced before they are saved in the `Fraction` object.

Copy constructor and assignment operator should work as expected:

```
Fraction f0(1, 2);
Fraction f1(f0); // copy constructor
f1 = f0;         // assignment operator
```

(Do you need to overwrite the default copy constructor and assignment operator for this class?)

Various unary and binary operators should work as expected:

```
Fraction f0(1, 2), f1(3, 4), f2;
f2 = +f0;
f2 = -f0;
f2 = (f0 += f1);
f2 = (f0 -= f1);
f2 = (f0 *= f1);
f2 = (f0 /= f1);
f2 = f0 + f1;
f2 = f0 - f1;
f2 = f0 * f1;
f2 = f0 / f1;
```

The `Fraction` class works seamlessly with integers:

```
Fraction f0(1, 2), f1(3, 4), f2;
f2 = f0 + 42;
f2 = f0 - 42;
f2 = f0 * 42;
f2 = f0 / 42;

f2 = 42 + f0;
f2 = 42 - f0;
f2 = 42 * f0;
f2 = 42 / f0;

f2 = (f0 += 42);
f2 = (f0 -= 42);
f2 = (f0 *= 42);
f2 = (f0 /= 42);

int i = 42, j;
j = (i += f0);
j = (i -= f0);
j = (i *= f0);
j = (i /= f0);
```

Comparisons are possible:

```
Fraction f0(-1, 2), f1(3, 4);
std::cout << (f0 == f1) << '\n';
std::cout << (f0 != f1) << '\n';
std::cout << (f0 < f1) << '\n';
std::cout << (f0 <= f1) << '\n';
std::cout << (f0 > f1) << '\n';
std::cout << (f0 >= f1) << '\n';
```

Finally it's possible to compute the absolute value and integer power of a `Fraction`:

```
Fraction f0(-1, 2);
std::cout << abs(f0) << '\n';
std::cout << pow(f0, 5) << '\n';
std::cout << pow(f0, -5) << '\n';
```

Review of pointers

The following is a quick review of pointers. Refer to my pointer notes for details.

Declaration: Suppose `T` is a type (example: `int`, `double`, `char`, `bool`, some struct, some class), then you do the following to declare `p` to be a pointer to a `T` value:

```
T * p;
```

At this point, `p` does not point to a valid value (whether on the heap on in a scope/frame) yet.

Allocating/using/deallocating a single value: To allocate a single `T` value for `p` to point to (or we can also say "to allocate a value for `p`"), we do

```
p = new T;
```

After this you can use the value `p` points to. You refer to this value as `*p`.

Note that the *address* of this value is stored in `p`. The *value* that `p` points to is `*p`. Make sure you see the difference between `p` and `*p`.

When you're done using this value, you can release this memory, or we say deallocate the value that `p` point to (or just say deallocate `p`), by doing

```
delete p;
```

Here's a simple example:

```
#include <iostream>

int main()
{
    double * p = new double;
    std::cout << "enter a double: ";
    std::cin >> (*p);

    std::cout << "you entered " << (*p) << std::endl;
    delete p;

    return 0;
}
```

Allocating/using/deallocating an array of values: Now suppose instead of a single value of type `T` you want to use an array of values of type `T`, say `n` such values. To allocate an array of `n` `T`–values for `p`, you do this:

```
p = new T[n];
```

After this, `p` points to the first of `n` `T`–values. The `i`–th *value* of the array `p` points to

is just `p[i]`. The *address* of the `i`–th value is `p+i`. Therefore the `i`–th value is also known as `*(p+i)`, i.e., `p[i]` and `*(p+i)` mean the same thing.

After you're done with this array of `T` values, you can deallocate the memory that `p` points to (i.e. release the memory back to the heap) if you do

```
delete [] p;
```

Here's a simple example:

```
#include <iostream>

int main()
{
    int n;
    std::cout << "how many double do you need? ";
    std::cin >> n;

    double * p = new double[n];
    for (int i = 0; i < n; i++)
    {
        std::cout << "enter double #" << i << ": ";
        std::cin >> p[i];
    }

    std::cout << "here are all your doubles: ";
    for (int i = 0; i < n; i++)
    {
        std::cout << p[i] << ' ';
    }
    std::cout << std::endl;

    delete [] p;

    return 0;
}
```

The above iterates over the value that `p` points to using index values. You can iterate using a pointer like this:

```
#include <iostream>

int main()
{
    int n;
    std::cout << "how many double do you need? ";
    std::cin >> n;

    double * p = new double[n];
    for (int i = 0; i < n; i++)
    {
        std::cout << "enter double #" << i << ": ";
        std::cin >> p[i];
    }

    std::cout << "here are all your doubles: ";
    for (double * q = p; q < p + n; ++q)
    {
        std::cout << (*q) << ' ';
    }
    std::cout << std::endl;

    delete [] p;

    return 0;
}
```

In this case, the pointer `q` (in the second `for`–loop) iteratively points to the values of the array that `p` points to.

Functions: Of course pointers, like any variable, can be passed to functions. So of course you can rewrite:

```
#include <iostream>

int main()
{
    int n;
    std::cout << "how many double do you need? ";
    std::cin >> n;

    double * p = new double[n];
    for (int i = 0; i < n; i++)
    {
        std::cout << "enter double #" << i << ": ";
        std::cin >> p[i];
    }

    std::cout << "here are all your doubles: ";
    for (int i = 0; i < n; i++)
    {
        std::cout << p[i] << ' ';
    }
    std::cout << std::endl;

    delete [] p;

    return 0;
}
```

as

```
#include <iostream>

void fill_doubles(double * p, int n)
{
    for (int i = 0; i < n; ++i)
    {
        std::cout << "enter double #" << i << ": ";
        std::cin >> p[i];
    }
}

void print_doubles(double * start, double * end)
{
    std::cout << "here are all your doubles: ";
    for (double * q = start; q < end; ++q)
    {
        std::cout << (*q) << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    int n;
    std::cout << "how many double do you need? ";
    std::cin >> n;

    double * p = new double[n];

    fill_doubles(p, n);
    print_doubles(p, p + n);

    delete [] p;

    return 0;
}
```

Recall that there's a memory debugging tool (google's asan – address sanitizer) that can check your code for memory problems. See class website or my website.

Q6. [Review of template class in CISS245 – dynamic arrays]

This is a review of writing a class template with a pointer and dynamic arrays in the heap.

The goal is to write a class to make C++ arrays a lot easier to use.

Specifically, the `vector` class is a class template that provides most of C++'s array features. Study the skeleton code for `vector` carefully. Specifically, note that each `vector` object has three members: `size_`, `capacity_` and `p_`. Obviously `p_` points to an array allocated in the free store, `capacity_` records the number of elements in the array, and `size_` records the number of elements in the array that are "in use". The file `test.cpp` includes some usage examples. `vector` is similar to our `IntDynArray` from CISS245 except that `IntDynArray` is not a class template. `vector` is similiar to the C++ STL `std::vector` class template.

Your goal is to complete the given skeleton file `vector.h`. You are advised strongly to test your code. In particular, you should add more test cases to `test.cpp`; what I've provided is absolutely not enough. If you work for a really good company, you will see that in some cases the test code can be longer than the code being tested.

It should be clear what you need to complete just by looking at the skeleton code. All methods mentioned in the skeleton code must be completed. (For your own learning, you may add other methods if you like.) Let me just mention a few things.

`vector` objects can be constructed by specifying a size:

```
vector< int > x(5);    // x models an int array of size 5
vector< double > y(4); // y models a double array of size 4
vector< char > x(5);   // x models a char array of size 3
```

The memory used is of course from the free store (or memory heap). Specifically, pointer `x.p_` will point to an array of 5 integer in the free store and `x.size_` `x.capacity_` is set to 5.

After the above you can read the values of "array" `x` (or rather the values of the array that `x.p_` points to):

```
std::cout << x[2] << '\n';
```

or you can write to the array:

```
x[2] = 42;
```

This means that your class must have `operator[]` for reading and writing. Since `vector` objects model arrays in C/C++, this is not surprising. However your `operator[]` must provide protection against invalid index values. This means that if your `vector`

object has size 5, then calling `operator[]` outside of the range of 0,1,2,3,4 will result in `IndexError` object being thrown as an exception object. The regular C/C++ arrays do not have this protection and is a very common programming error resulting not just in programs crashing but in fact can cause serious security issues.

Of course there's the destructor, copy constructor, and assignment operator. Once you have memory allocation in your constructor(s), you must overwrite these, right?

Another important constructor is

```
vector(T arr[], int size);
```

This allows you to construct a `vector` object from a regular C/C++ array. For instance the above constructor allows you to do the following

```
int x[] = {1,3,5,2,4,6};
vector< int > y(x, 6);
```

Otherwise you would have to do this:

```
int x[] = {1,3,5,2,4,6};
vector< int > y(6);
for (int i = 0; i < 6; i++)
{
    x[i] = y[i];
}
```

Code to print the vector is provided so that you can do this:

```
std::cout << x << '\n';
```

Study it carefully. By the way, printing like the above is probably among the earliest thing to write. How in the world do you debug if you can't print?!? (Also, see the note below on debug printing.)

The comparison operator, `operator==`, returns `true` exactly when the two relevant `vector` objects model the same array, i.e., if you call

```
x == y
```

where `x` and `y` are, say `vector< char >` objects, then you get `true` exactly when their sizes are the same and the values in the arrays that they point to have the same values. This operator must also be coded very early in order to automate testing.

Of course the assignment operator, `operator=()`, does the obvious. If you have

```
vector< double > x(5);
vector< double > y(10);
// code to put values into the arrays of x and y
```

then on calling the assignment operator:

```
x = y;
```

the arrays (in the free store) of both `x` and `y` have the same values; of course the size of `x` and `y` becomes the same.

All the rest should be obvious from reading the skeleton code.

The above is what you would expect to make C++'s array more user-friendly.

Now for the `insert` method. This is not directly available in your usual C/C++ arrays.

Add a method `insert(int index, const T & value)` so that you can insert a value into the array at the given index position. This is the expected behavior (the following includes 3 test cases):

```
vector< int > a(3);
a[0] = 5;
a[1] = 10;
a[2] = -6;
std::cout << a << std::endl; // prints [5, 10, -6]

a.insert(0, 42);
std::cout << a << std::endl; // prints [42, 5, 10, -6]

a.insert(2, 999);
std::cout << a << std::endl; // prints [42, 5, 999, 10, -6]

a.insert(5, -11111);
std::cout << a << std::endl; // prints [42, 5, 999, 10, -6, -11111]
```

Note that if `a` has size 5, you can only call `insert` for `index = 0, 1, 2, 3, 4, 5`. You cannot for instance insert at index value -1 and you cannot insert at index value 6. In general if the size of `a` is `s`, then you can only insert at `index = 0, 1, 2, ..., s`. If `insert` is invoked at index outside this range your code should throw an `IndexError` exception object.

Here are test cases that test correct index positions:

```
try
{
    vector< int > a(10);
    a.insert(0, 42);
    std::cout << "pass\n";
}
catch (IndexError & e)
{
```

```
    std::cout << "fail\n";
}

try
{
    vector< int > a(10);
    a.insert(5, 42);
    std::cout << "pass\n";
}
catch (IndexError & e)
{
    std::cout << "fail\n";
}

try
{
    vector< int > a(10);
    a.insert(10, 42);
    std::cout << "pass\n";
}
catch (IndexError & e)
{
    std::cout << "fail\n";
}

try
{
    vector< int > a(10);
    a.insert(10, 42);
    std::cout << "pass\n";
}
catch (IndexError & e)
{
    std::cout << "fail\n";
}
```

Here are some test cases that test invalid index positions:

```
try
{
    vector< int > a(10);
    a.insert(-6, 42);
    std::cout << "fail\n";
}
catch (IndexError & e)
{
    std::cout << "pass\n";
}

try
```

```
{
    vector< int > a(10);
    a.insert(20, 42);
    std::cout << "fail\n";
}
catch (IndexError & e)
{
    std::cout << "pass\n";
}
```

Add a method to concatenate two `vector` objects. This involves overloading the `operator+=` and `operator+`. For instance if `a` and `b` are `vector< int >` objects, then `a += b` will result in the values of `b` being appended to `a` (in the obvious way). Note that `operator+=` should return a reference to the object invoking the operator. In other words, your class should allow you to do this:

```
(a += b) += b;
```

The `push_back` method adds a value to the end of the array that your `vector` models.

```
int x[] = {1, 3, 5, 2, 4, 6};
vector< int > y(x, 6); // y is {1, 3, 5, 2, 4, 6}
y.push_back(42);       // y is {1, 3, 5, 2, 4, 6, 42}
```

Note that the `y.size_` is changed from `6` to `7`. If the array that `y.p_` cannot accomodate 7 values (i.e., if `y.capacity_` is 6), then you need to reallocate a larger array for `y.p_` to point to (just like the `IntDynArray` example/assignment from CISS245). In this case, when you reallocate, the array must be twice of what you need. For instance in the above example, since `y.p_` needs 7 values, the new array allocated must have 14 values.

The skeleton code is of course not correct. You have to work on it. I do not need to tell you this but I'm going to anyway: Make sure that you use pass-by-reference for object parameters as much as possible. Also, parameters and methods should be constant whenever possible.

(The `IntDynArray` example/assignment from CISS245 and also the C++ STL `std::vector` has more methods than described above. You need not implement these extra methods.)

Of course you have must ensure that there are no memory issues such as memory leaks and double free errors. You should definitely check your code with google asan.

Note on debug printing

Frequently, it is helpful to have the ability to print everything useful for debugging purposes. Our `operator<<` is only helpful when *using* this `vector` class and debugging at a higher level assuming that the `vector` class is already working perfectly. However for debugging `vector`, besides printing the values in the array, it might be helpful to print the `size` and the pointer `p` as well. I suggest you write a method

```
void debugprint() const;
```

to print everything associated to the object, i.e., print the `size` and `p` besides the array values. This is very helpful especially in chasing pointers. In more complex classes with pointers, you absolutely have to print the pointers for debugging. This method is not required for this assignment.

Skeleton for `vector.h`:

```
// File: vector.h

#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>
#include <string>

class IndexError
{};


template < typename T >
class vector
{
public:
    vector(int size)
        : size_(size), capacity_(size), p_(new T[size])
    {}

    // TODO: Set attributes, copy values at x.p_ to p_
    vector(const vector & x)
        : size_(0), p_(new T[0])
    {
        for (int i = 0; i < x.size_; ++i)
        {
            p_[i] = (x.p_)[i];
        }
    }

    vector(T arr[], int n)
    {
    }

    ~vector()
    {
        // TODO: deallocate memory used by p
    }

    int size() const
    {
        return size_;
    }

    int capacity() const
    {
        return capacity_;
    }
```

```
    T operator[](int index) const
    {
        if (index < 0 || index > size_)
        {
            throw IndexError();
        }
        // FIXIT
        return p_[0];
    }

    T & operator[](int index)
    {
        // FIXIT
        return p_[0];
    }

    const vector & operator=(const vector & x)
    {
        if (this != &x)
        {
            // FIXIT: Similar to copy constructor
        }
        return (*this);
    }

    bool operator==(const vector & x) const
    {
        // FIXIT
        return true;
    }

    void insert(int index, const T & value)
    {
        // TODO
    }

    void push_back(const T & value)
    {
        // TODO
    }

private:
    int size_;
    int capacity_;
    T * p_;
};


template < typename T >
std::ostream & operator<<(std::ostream & cout,
                          const vector< T > & x)
```

```
{
    int size = x.size();
    cout << '{';
    std::string delim = "";
    for (int i = 0; i < size; ++i)
    {
        cout << delim << x[i]; // Recall: x[i] calls x.operator[](i)
        delim = ", ";
    }
    cout << '}';
    return cout;
}

#endif
```

Minimal `test.cpp`:

```
// File: test.cpp

#include <iostream>
#include "vector.h"

int main()
{
    std::cout << "creating vector< int > object a ...\n";
    vector< int > a(5);
    a[0] = 42; // a.operator[](0) = 42
    a[1] = 0;
    a[2] = 0;
    a[3] = 0;
    a[4] = -5000;
    std::cout << "a: " << a << std::endl;

    std::cout << "testing exception catching ...\n";
    try
    {
        std::cout << a[6] << '\n';
    }
    catch (IndexError & e)
    {
        std::cout << "successfully caught IndexError exception\n";
    }

    /*

    vector< int > b(3);
    b = a; // b.operator=(a)
    std::cout << b << std::endl;

    vector< int > c(0), d(0);
    c = d = b;
    std::cout << c << '\n';
    std::cout << d << '\n';

    vector< int > e(3);
    for (int i = 0; i < 3; ++i)
    {
        e[i] = i;
    }
    std::cout << "e : " << e << std::endl;

    e.push_back(42);
    std::cout << "e : " << e << std::endl;

    */
```

```
    /*
    std::cout << "creating vector< double > object x ...\n";
    vector< double > x(10);
    for (int i = 0; i < 10; i++)
    {
        x[i] = 0.1 * i;
    }
    std::cout << "x: " << x << std::endl;
    */

    /*
    std::cout << "creating vector< char > object y ...\n";
    vector< char > y(26);
    for (int i = 0; i < 26; i++)
    {
        y[i] = 'A' + i;
    }
    std::cout << "y: " << y << std::endl;
    */

    return 0;
}
```

Q7. [Review of template class in CISS245 – pointer class]

This is a review of writing a template class with a pointer that points to a single value.

Write a template pointer class, `ptr`. (The previous problem is an array class with a pointer that points to an array of values. For this problem, the pointer points to a single value.)

The header file is named `ptr.h`. Each object of `ptr` contains a pointer that when allocated points to *one* value of type `T` where `T` is the template type parameter of the `ptr` class. Call the pointer `p_`.

Here are some usage examples illustrating what your class can do:

```
void f()
{
    ptr< int > p;    // p.p_ points to an int value in the heap
    ptr< double > q; // q.p_ points to a double value in the heap
    ptr< char > r;   // r.p_ points to a char value in the heap

    *p = 42;
    std::cout << (*p) << std::endl; // prints 42
    *p = 43;
    std::cout << (*p) << std::endl; // prints 43

    ptr< int > s(42); // here's another constructor. s.p_ will point to
                      // a 42 in the heap.

    // here ... where p,q,r,s go out of scope, their destructor will
    // deallocate memory used by their pointers.
}
```

Clearly you must overwrite the default destructor, copy constructor, and assignment operator of `ptr`.

```
void g()
{
    ptr< int > p(42);
    ptr< int > q(p);  // q.p_ points to 42 in the heap which is different
                      // from the 42 that p.p_ points to
    ptr< int > r;
    r = p;            // the integer that r.p_ points to is changed to 42
}
```

Of course you have must ensure that there is no memory issues such as memory leaks and double free errors. You should definitely check your code with google asan.