1. Explore the class IntStack discussed in lecture notes. Implement the following missing operations for IntStack. What other additional operations should a practical container class provide? int pop (); // remove and return the top value IntStack& operator = (IntStack const&); // assignment (value copy) If possible, compile and test your code.

Answer:

I defined the pop function in cpp file.

```
int IntStack::pop() {
    if (top_ == 0) {
        throw std::logic_error("nothing in stack");
    }
    else {
        top_--;
    }
}
```

I defined the operator = overloaded function in the header file directly

```
IntStack& operator = (IntStack const& other) {
    IntStack ret;
    for (size_t i = 0; i < top_; ++i) ret.array_[i] = other.array_[i];
}
```

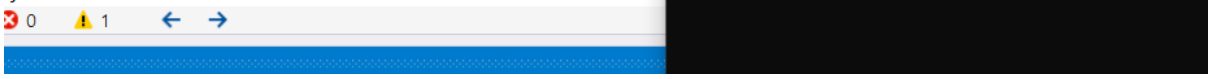
I also have a print() function to print out the stack.

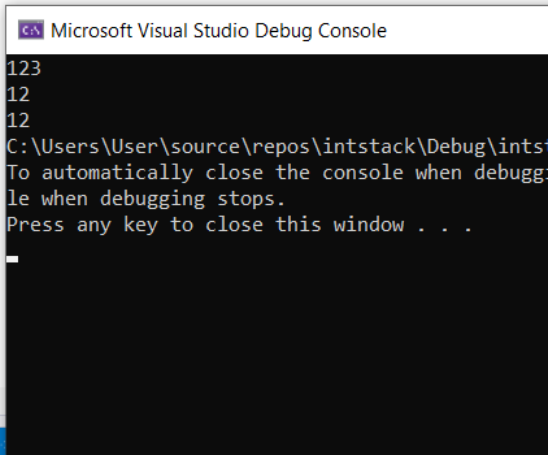
```
int IntStack::print() {
    if (top_ == 0) {
        throw std::logic_error("nothing in stack");
    }
    else {
        for (size_t i = 0; i < top_; i++) {
            std::cout << array_[i];
        }
    }
}
```

Result

```
int main() {
    IntStack a;
    a.push(1);
    a.push(2);
    a.push(3);
    a.print();
    std::cout<<std::endl;
    a.pop();
    a.print();
    std::cout << std::endl;
    IntStack b = a;
    b.print();
    return 0;

}
```





2. Why bother about classes and object-oriented programming and language features when you can well express any needed computation with functions only - like in C?

Answer: As the program gets bigger, object-oriented programming and classes would encourage the programmer to place data where it is not directly accessible by the rest of the program, unlike in C where we only have functions in which we would need to expose our program's data to be accessible from any part of the program, allowing any function to modify any piece of data might potentially cause bugs. By using OOP, we are only allowing the methods inherited by the class objects to retrieve or modify those data. Essentially, it allows the codes to be more maintainable, reusable and scalable to big project.

3. (a) Discuss benefits and problems or dangers related to overloading in C++. Explore the overloading of function names and operator symbols in C++. Why is it needed and how does it work? What can be overloaded? What operators or operations cannot be overloaded? What are the potential drawbacks or dangers related to using overloading in C++ programming? Give short illustrative examples. Explain your examples.

Answer:

Overloading allows us to specify more than one definition for a function name or an operator in the same scope. It is useful to ensure that the code is clean and easy to understand by not having to use different function names for same kind of operations. The overloaded functions must have different types of arguments or a different number of arguments so that the overload resolution can differentiate between the different functions of the same name. If we have defined our own data types and we want to allow operation such as addition to our data type, we can have operator overloading to achieve that. Most operators can be overloaded except for . (dot) :: ?: sizeof

Below are more functions that cannot be overloaded.

Function declarations that differ only by its return type cannot be overloaded with function overloading process because the compiler can't differentiate which function to use.

Void func();

Int func();

Function declarations with the same parameters or the same name types cannot be overloaded if any one of them is declared as a static function.

E.g.

```
class X{
        static void func();
        void func(); // Does not work
};
```

Function with default arguments

Say we have two functions

int func (int I, int y = 12){}

int func (int x){}

They can both be called by using one argument which would throw a function (int) is ambiguous error.

The drawback of overloading is that if we are trying to be very flexible and account for all of the niche cases it might make some of the functions to rarely or never be used at all and would just make the program unnecessarily complex.

(b) Explore the C++ standard library iterator functionality. Explain (again) the main idea behind the Iterator pattern: why are they needed and useful? How are iterators related to the C++ standard containers? In what manner do C++ iterators utilize overloading? Explain why and how. How safe are C++ iterators, generally?

Answer:

An iterator is an object that can iterate over elements in a C++ standard library container and provide access to individual elements without having to be concerned with the type of container the elements are stored in. It consists of two operations begin () and end() to return the beginning position of the STL container and the end position of the container so that it can be iterated by incrementing the ptr to point from the start of the position to the end of the position.

E.g. we have an integer vector

```
vector<int> ar = { 1, 2, 3, 4, 5 };
```

//We can declare iterator to the vector by

```
vector<int>::iterator ptr;
```

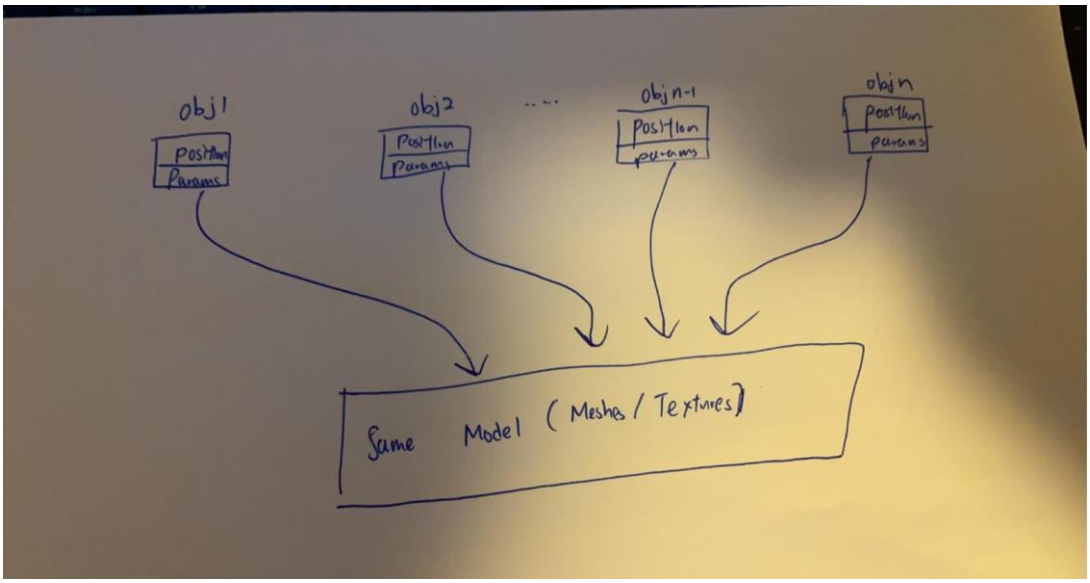
// Displaying vector elements using begin() and end()

```
cout << "The vector elements are : ";
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
   cout << *ptr << " ";
```

The c++ iterators are not safe in the sense that you can increment an iterator past the end, which will result in undefined behaviour.

4. Explore the Flyweight design pattern as a game programming pattern. Explain its general purpose, structure, and consequences. For what situations or purposes would Flyweight be used in game programs? Sketch a sample design for Flyweight. Implement it in a suitable pseudolanguage and explain verbally what is needed to make it work. Discuss the other (game) programming patterns related to this one. How do they resemble or potentially use the Flyweight pattern?

Answer: The general idea is to minimize the amount of data we have to push to the GPU, thus we would have a shared data just once and every object that shares the specific shared data would point to the data with its own unique data such as its position, colour and scale to render out the objects. For example, trees that have different coordinates and params might share the same meshes/textures of the bark/branches/leaves and we just need to know its specific params and point to the shared data to render out the tree. Below is an illustration of the example,



Say we have a shared texture class, we just need to differ our object by its position/height/thickness to create the unique objects rendered out by the shared texture.

```
Class SharedTexture
{
Private:
        Mesh mesh_;
        Texture texture_;
}
Class Object
{
Private:
        SharedTexture* model_;
        Vector position_;
        Double height_;
```

```
    Double thickness_;
}
```

This is related to Factory Method pattern in which we encapsulate the construction of the object behind some interface that can first look for an existing object.

In order to return a previously created flyweight, we can keep track of the pool of ones that were already instantiated. Hence, we can use object pool to keep track of the flyweight data.