8. (8 points) Merlin was showing off his operator overloading wizardry to his roomies. He asked them to "*check out these lines of code*":

```
vector<int> vect;
vect << 2;
```

Figure 1: Code

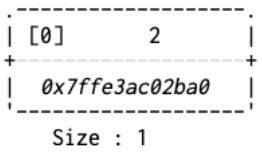


Figure 2: Effect on `vect`

And just like that, Merlin added the integer value `2` to `vector<int> vect`. Shouts of glee could be heard from outside, while fist bumps were prevalent inside. He wasn't through yet though, and showed off once more:

operator <<

vect << 7 << 11

```
vect << 7 << 11;
```

Figure 3: Code

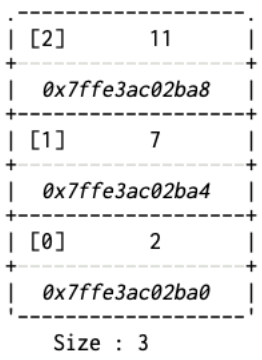


Figure 4: Effect on `vect`

Having added two more values to `vect` using `operator<<`, the group's excitement was heard across all of Aggieland. He then challenged his friends to define and declare, an overloaded operator that does that exhibited here.

Write the declaration and definition for an overloaded `operator<<` that adds an `int` to the back of a `vector<int>`. Make sure that your definition ensures that expressions such as `vect << 2` and `vect << 7 << 11` evaluate correctly.

Function Declaration

```
vector<int>& operator<<(vector<int>&, int);
```

Function Definition

```
vector<int>& operator<<(vector<int>& v, int i)
{
    v.push_back(i);
    return v;
}
```

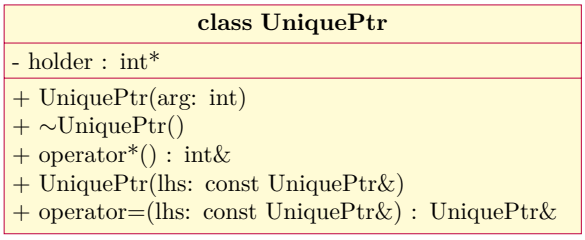
7. In this problem you will implement a simple 'smart pointer'. The objects derived from this 'blueprint' will maintain a unique pointer to a dynamically allocated `int` object storing an intger value. The constructor for objects of this type will be responsible for:

   - creating a dynamically allocated object of type `int` with the new operator,
   - initializing that dynamically allocated object with the passed value, and
   - storing the address to that object in a data member.

   When a 'smart pointer' object's lifetime is up, its destructor will be called. The destructor will be responsible for:

   - freeing the dynamic memory obtained by the constructor.

   With `UnqiuePtr`, we need not worry about memory management of dynamically allocated objects through `new` and `delete`, as we've created a type that manages this for us. The structure of `UnqiuePtr` is detailed in the following UML diagram:



   (a) (5 points) Write the class definition for the `UniquePtr` class.

```
class UniquePtr {
public:
    UniquePtr(int);
    ~UniquePtr();
    int& operator*();
private:
    int *holder;
};
```

uniquePtr up(11);

(b) (5 points) Define the parameterized constructor for `UniquePtr` that uses the value stored in its parameter `arg` to initialize `holder` with the address of a dynamically allocated object of type `int` storing `arg`.

uniquePtr::uniquePtr(int arg) : holder(nullptr)
{
    holder = new int(arg);
}

```
holder = new int(arg); // NOT int[arg]
```

(c) (5 points) Define the destructor for `UniquePtr` that deallocates the memory associated with the dynamically allocated object pointed to by `holder`.

uniquePtr::~uniquePtr()
{
    delete holder;
}

(d) (5 points) Define the dereference operator (`operator*`) for `UniquePtr`. This operator should return a *reference to the object pointed to by* `holder`.

int& operator*()
{
    return *holder;
}

uniquePtr up(7);
*up = 8;

(e) (5 points) How could you inhibit an object of type `UniquePtr` from being initialized or assigned an object of the same type?