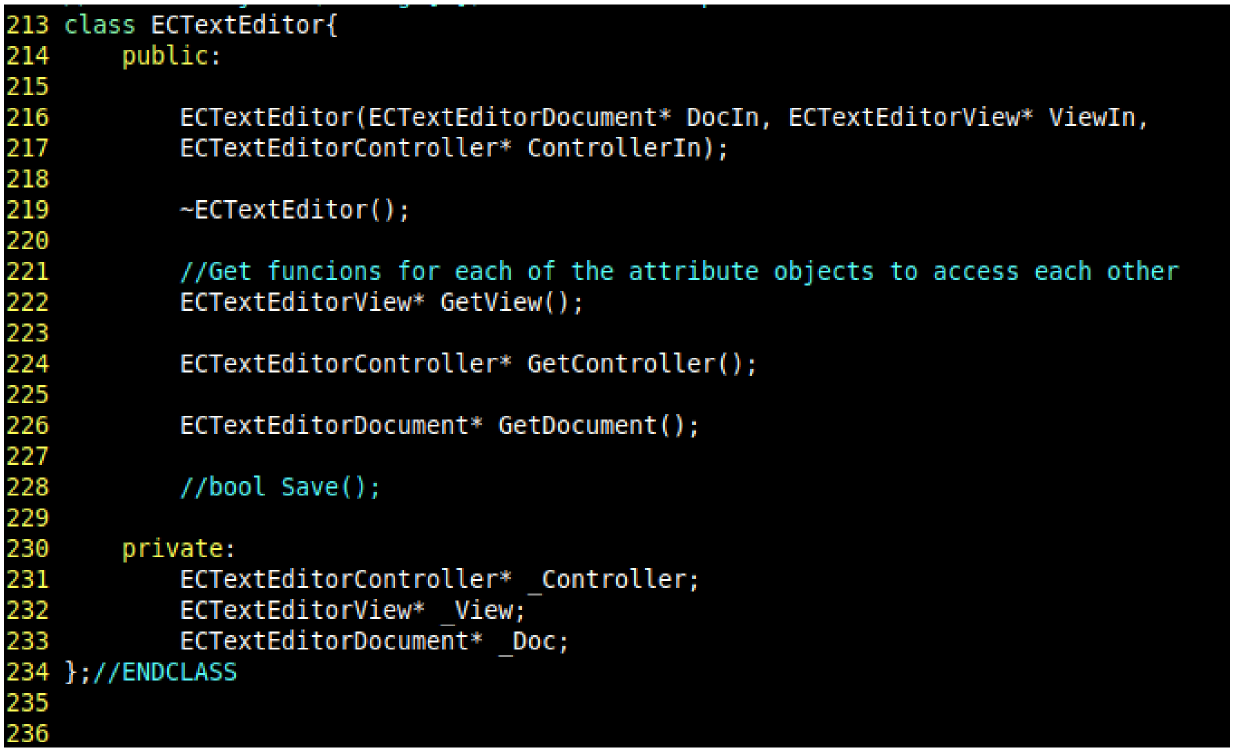
Erik O Kriz
Text Editor Write-up

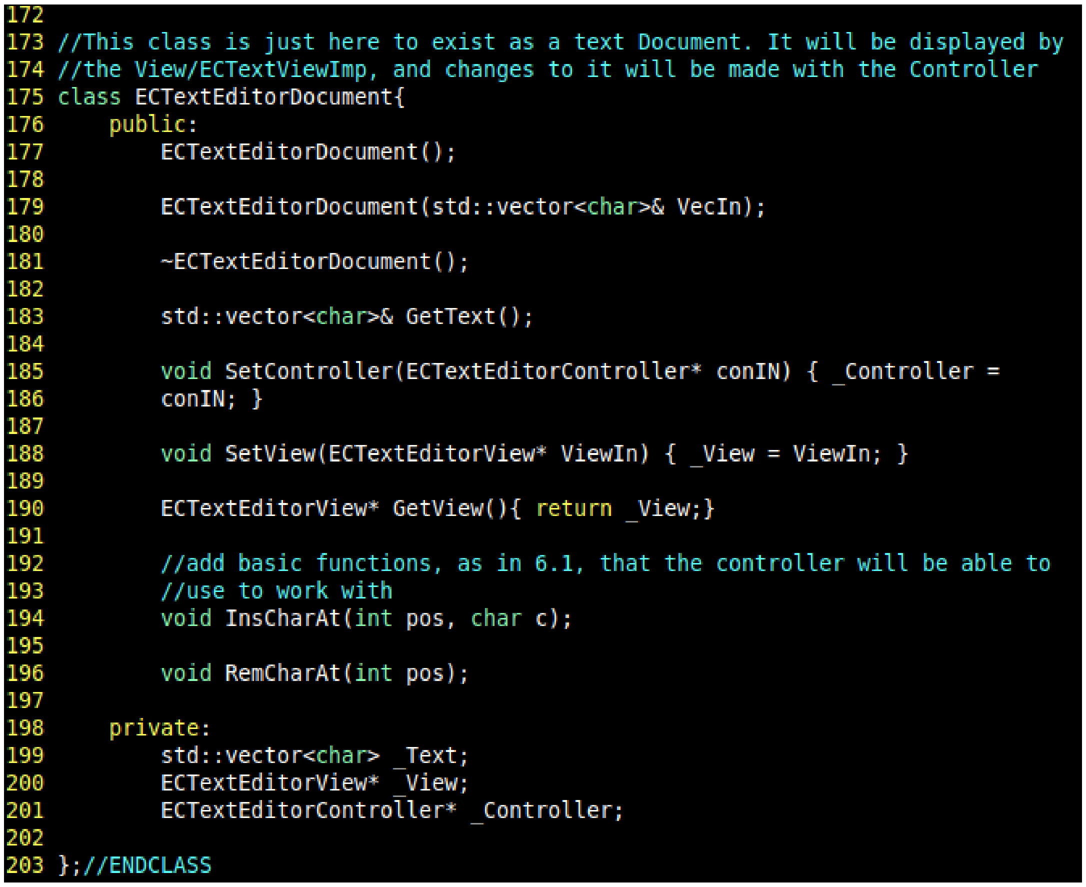
The core of this text editor is based on 3 complex class called the Controller, View, and Document. The class which is created upon running this code's executable is simply called the Text Editor. This class takes one of each of the core classes as inputs, and in its constructor, links all of the core classes together.



```
class ECTextEditor{
    public:

        ECTextEditor(ECTextEditorDocument* DocIn, ECTextEditorView* ViewIn,
        ECTextEditorController* ControllerIn);

        ~ECTextEditor();

        //Get funcions for each of the attribute objects to access each other
        ECTextEditorView* GetView();

        ECTextEditorController* GetController();

        ECTextEditorDocument* GetDocument();

        //bool Save();

    private:
        ECTextEditorController* _Controller;
        ECTextEditorView* _View;
        ECTextEditorDocument* _Doc;
};//ENDCLASS
```

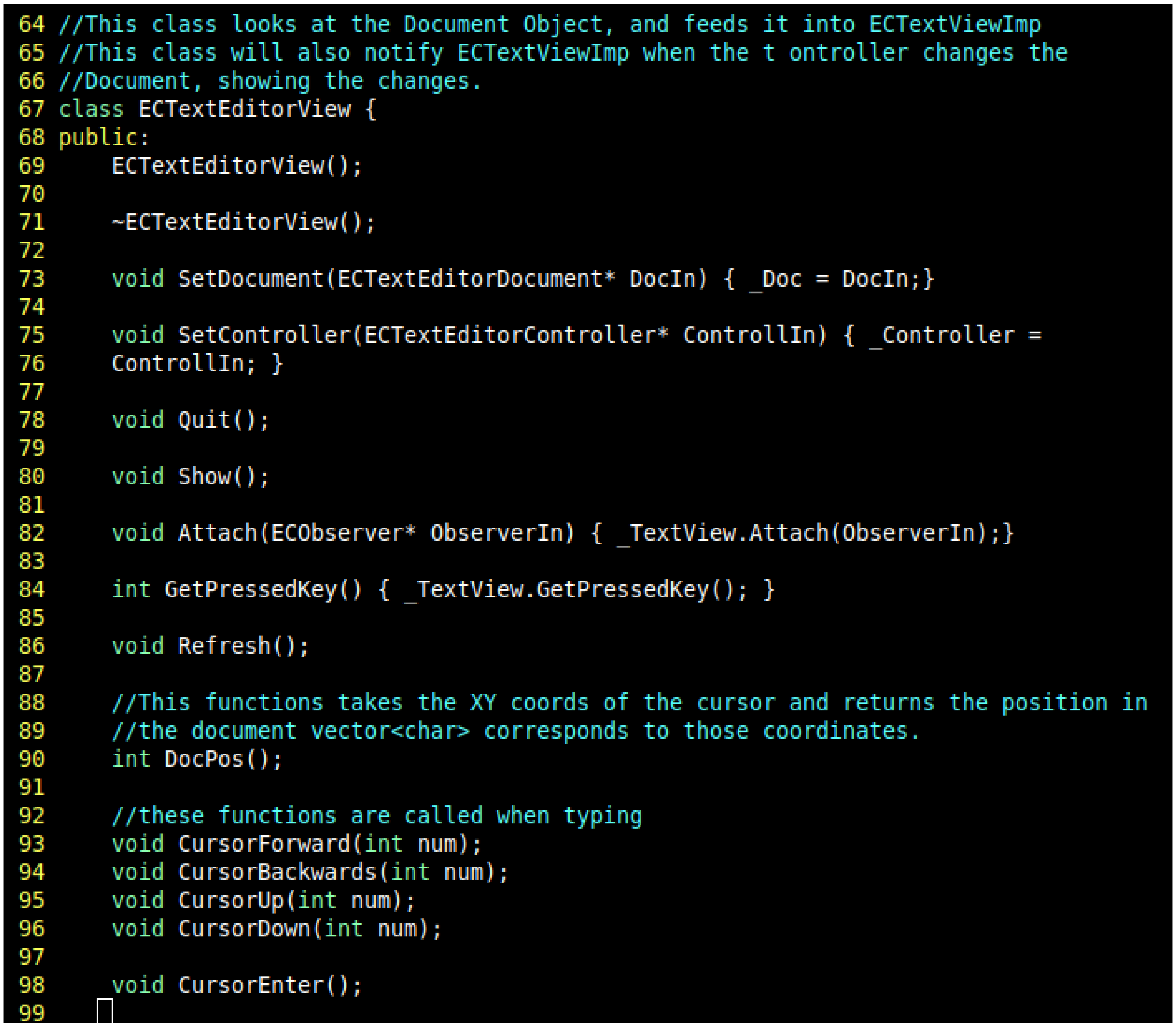
- Here you can see the main Text Editor class, it has a pointer to each of the 3 core classes, and requires a pointer to one of each class in order to be created and link them all together.

Document: The Document class is the simplest out of the 3 core classes. Its main function is to hold the text data we're manipulating as a vector of characters. Opening a text file loads all data within it into this vector. As the user types, this is the data which is changed. When the user then exits the text editor, this changed data is saved into the text file.



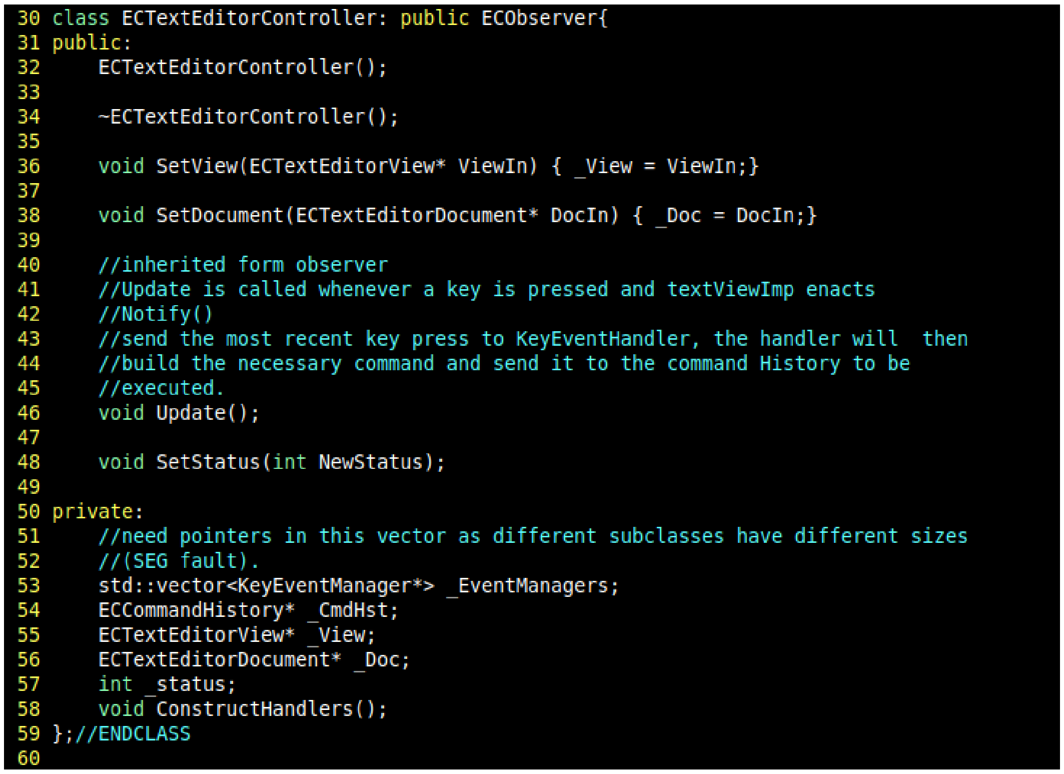
```cpp

//This class is just here to exist as a text Document. It will be displayed by
//the View/ECTextViewImp, and changes to it will be made with the Controller
class ECTextEditorDocument{
    public:
        ECTextEditorDocument();

        ECTextEditorDocument(std::vector<char>& VecIn);

        ~ECTextEditorDocument();

        std::vector<char>& GetText();

        void SetController(ECTextEditorController* conIN) { _Controller =
        conIN; }

        void SetView(ECTextEditorView* ViewIn) { _View = ViewIn; }

        ECTextEditorView* GetView(){ return _View;}

        //add basic functions, as in 6.1, that the controller will be able to
        //use to work with
        void InsCharAt(int pos, char c);

        void RemCharAt(int pos);

    private:
        std::vector<char> _Text;
        ECTextEditorView* _View;
        ECTextEditorController* _Controller;

};//ENDCLASS
```

View: The function of the View class is to display the text editor to the user. This is done in a page based format. Meaning you are shown as much text as your screen allows on the first page, and if you want to see later text you scroll past the bottom or use the page down key to see the next block of text. The View is also responsible for managing the placement of the cursor (while also keeping track of the place in the Document's character vector that cursor placement corresponds to), as well as displaying any relevant information.



```
//This class looks at the Document Object, and feeds it into ECTextViewImp
//This class will also notify ECTextViewImp when the t ontroller changes the
//Document, showing the changes.
class ECTextEditorView {
public:
    ECTextEditorView();

    ~ECTextEditorView();

    void SetDocument(ECTextEditorDocument* DocIn) { _Doc = DocIn;}

    void SetController(ECTextEditorController* ControllIn) { _Controller =
    ControllIn; }

    void Quit();

    void Show();

    void Attach(ECObserver* ObserverIn) { _TextView.Attach(ObserverIn);}

    int GetPressedKey() { _TextView.GetPressedKey(); }

    void Refresh();

    //This functions takes the XY coords of the cursor and returns the position in
    //the document vector<char> corresponds to those coordinates.
    int DocPos();

    //these functions are called when typing
    void CursorForward(int num);
    void CursorBackwards(int num);
    void CursorUp(int num);
    void CursorDown(int num);

    void CursorEnter();

```

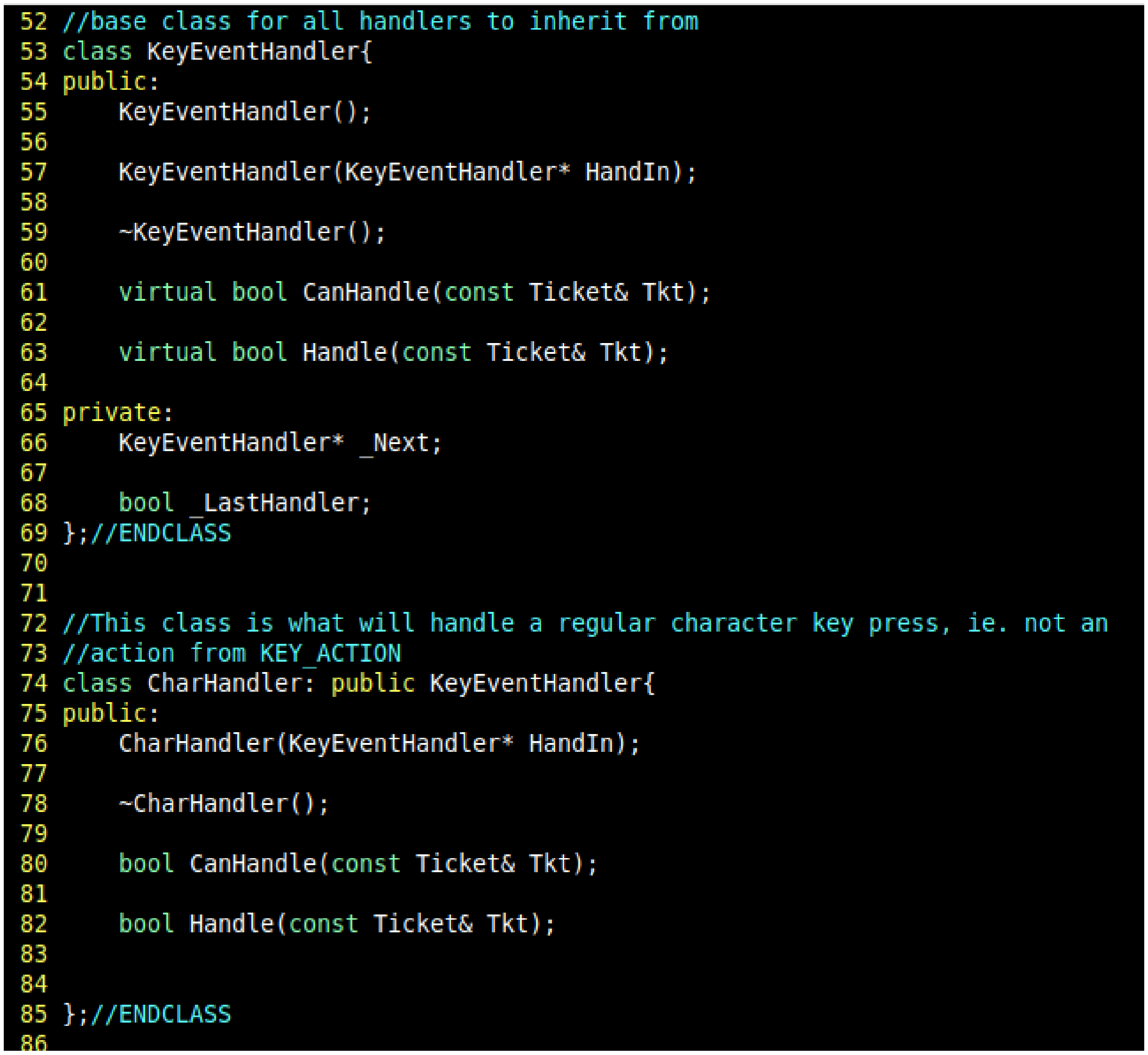
Controller: The purpose of this class is to send commands to the view based on user inputs. It inherits from the Observer class, which is part of the visual library this editor is based on. The purpose of the observer class is to be attached to the view, and whenever the user presses an input, the view calls the Update() functions on all observers which are attached to it. In the Controller's case, it passes this input on to a Chain of Responsibility construct in order to handle each individual key press.



```
class ECTextEditorController: public ECObserver{
public:
    ECTextEditorController();

    ~ECTextEditorController();

    void SetView(ECTextEditorView* ViewIn) { _View = ViewIn;}

    void SetDocument(ECTextEditorDocument* DocIn) { _Doc = DocIn;}

    //inherited form observer
    //Update is called whenever a key is pressed and textViewImp enacts
    //Notify()
    //send the most recent key press to KeyEventHandler, the handler will  then
    //build the necessary command and send it to the command History to be
    //executed.
    void Update();

    void SetStatus(int NewStatus);

private:
    //need pointers in this vector as different subclasses have different sizes
    //(SEG fault).
    std::vector<KeyEventManager*> _EventManagers;
    ECCommandHistory* _CmdHst;
    ECTextEditorView* _View;
    ECTextEditorDocument* _Doc;
    int _status;
    void ConstructHandlers();
};//ENDCLASS

```

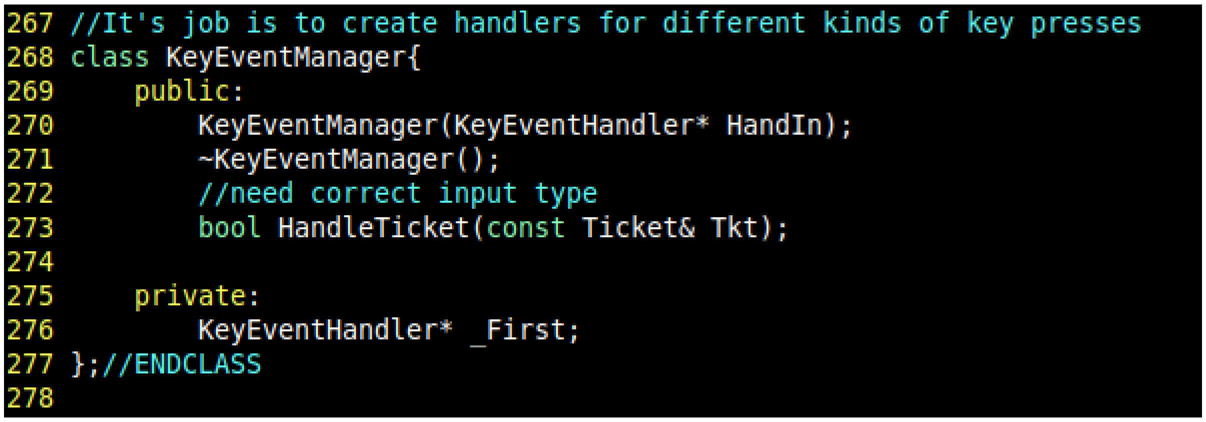
Chain of Responsibility: This is an abstract data structure whose job it is to handle all inputs from the user, sending commands to the View and Document in order to achieve it. There are 3 base classes which are involved in this chain, with many classes which inherit form them. The Handler class is one which other classes inherit from, the idea being that each different handler class handles a different input form the user. These handler sub-classes are connected in a linked list, each of them containing an input code which they are able to handle. If they cannot handle it they pass the input code on to the next object in the list. This goes on until one of the objects is able to handle the user input code, at which point it will send commands to the view, document, and the Command History (more on that later). An object which is a regular Handler class exists at the end of this list, it does not have any commands tied to it, it simply says it can handle anything and does nothing. It is there to handle any inputs that have not been coded yet or inputs that should do nothing.

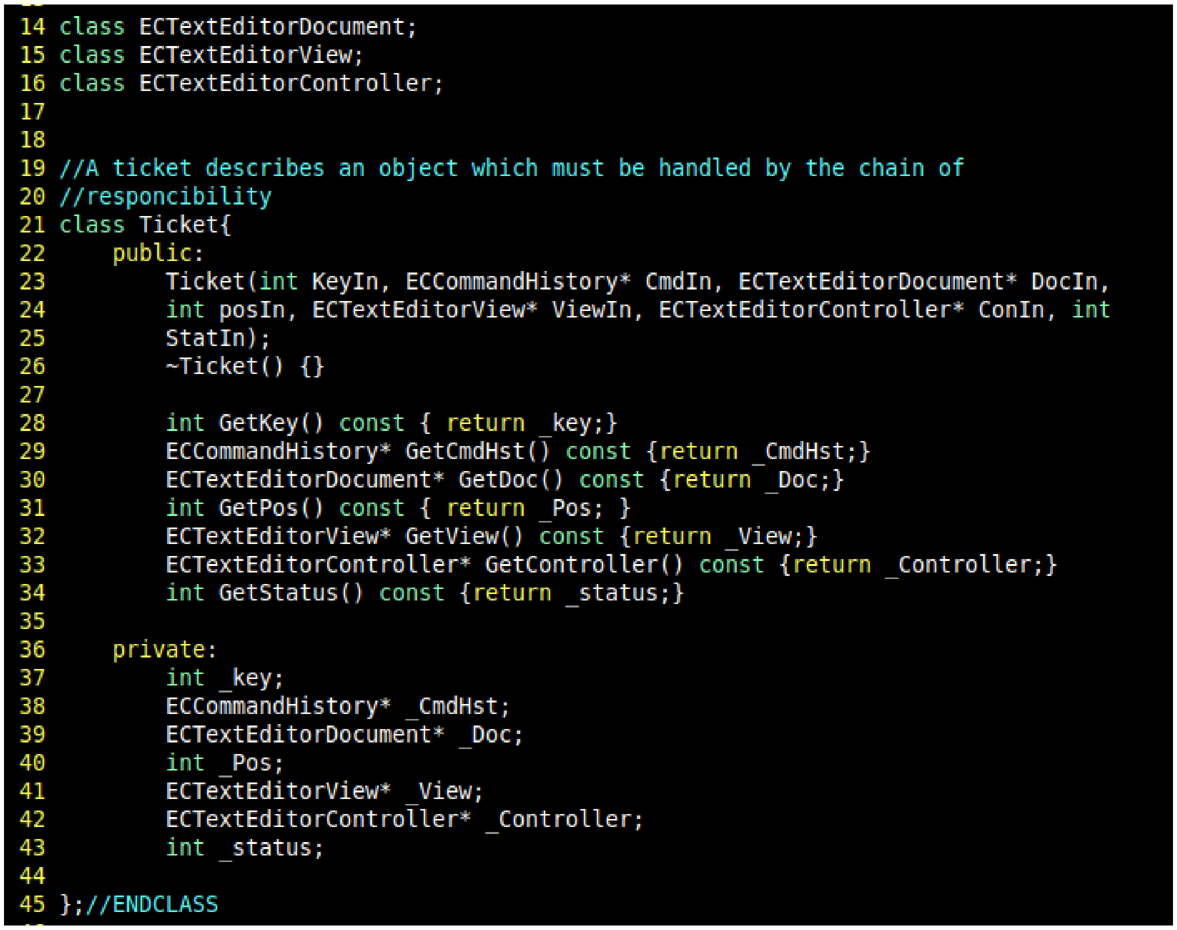
The Manager class is what the Controller class has a pointer to. Its job is to construct each of the handlers and order them into a linked list. When a user presses an input it creates an object of the third base class of this chain, a Ticket. Each of these tickets contain the code of the user input, the state of the view, and various pointers to help each handler send commands. This Ticket is what gets sent down the list of handlers until its input code is able to be handled.



```
//base class for all handlers to inherit from
class KeyEventHandler{
public:
    KeyEventHandler();

    KeyEventHandler(KeyEventHandler* HandIn);

    ~KeyEventHandler();

    virtual bool CanHandle(const Ticket& Tkt);

    virtual bool Handle(const Ticket& Tkt);

private:
    KeyEventHandler* _Next;

    bool _LastHandler;
};//ENDCLASS


//This class is what will handle a regular character key press, ie. not an
//action from KEY_ACTION
class CharHandler: public KeyEventHandler{
public:
    CharHandler(KeyEventHandler* HandIn);

    ~CharHandler();

    bool CanHandle(const Ticket& Tkt);

    bool Handle(const Ticket& Tkt);


};//ENDCLASS

```

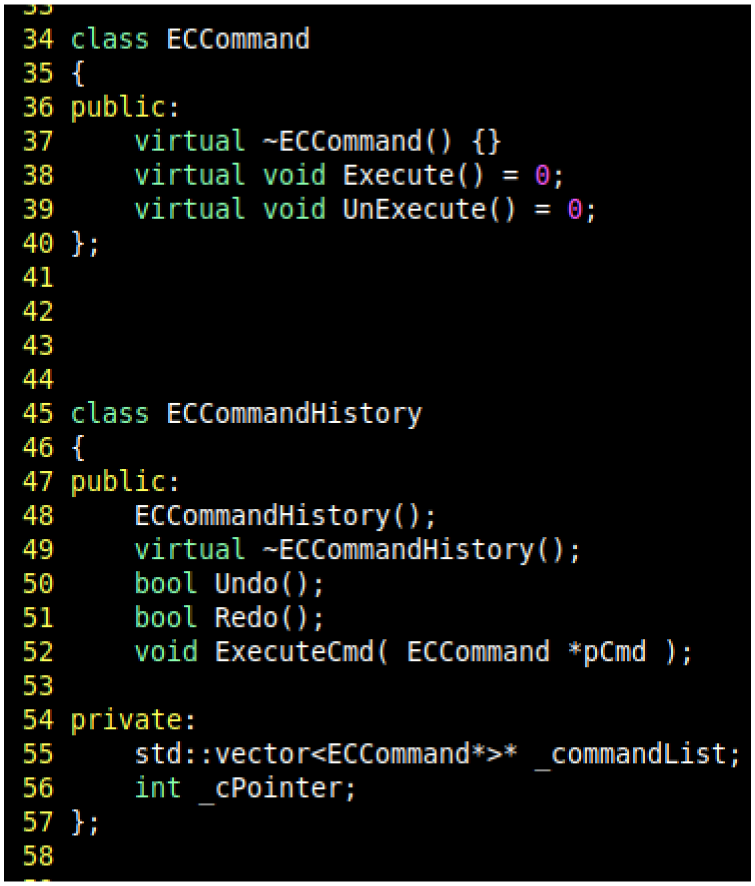
- Here you can see the base handler class as well as a sub-class.


```
//It's job is to create handlers for different kinds of key presses
class KeyEventManager{
    public:
        KeyEventManager(KeyEventHandler* HandIn);
        ~KeyEventManager();
        //need correct input type
        bool HandleTicket(const Ticket& Tkt);

    private:
        KeyEventHandler* _First;
};//ENDCLASS

```
- Here you can see the Manager class it is given a handler as the first spot in the linked list, and once it is created it only has a pointer to the first object in the list. Each subsequent handler has a pointer to the next handler. So all the manager has to do is to take the input Ticket and pass it on to the first handler


```
class ECTextEditorDocument;
class ECTextEditorView;
class ECTextEditorController;


//A ticket describes an object which must be handled by the chain of
//responcibility
class Ticket{
    public:
        Ticket(int KeyIn, ECCommandHistory* CmdIn, ECTextEditorDocument* DocIn,
        int posIn, ECTextEditorView* ViewIn, ECTextEditorController* ConIn, int
        StatIn);
        ~Ticket() {}

        int GetKey() const { return _key;}
        ECCommandHistory* GetCmdHst() const {return _CmdHst;}
        ECTextEditorDocument* GetDoc() const {return _Doc;}
        int GetPos() const { return _Pos; }
        ECTextEditorView* GetView() const {return _View;}
        ECTextEditorController* GetController() const {return _Controller;}
        int GetStatus() const {return _status;}

    private:
        int _key;
        ECCommandHistory* _CmdHst;
        ECTextEditorDocument* _Doc;
        int _Pos;
        ECTextEditorView* _View;
        ECTextEditorController* _Controller;
        int _status;

};//ENDCLASS
```

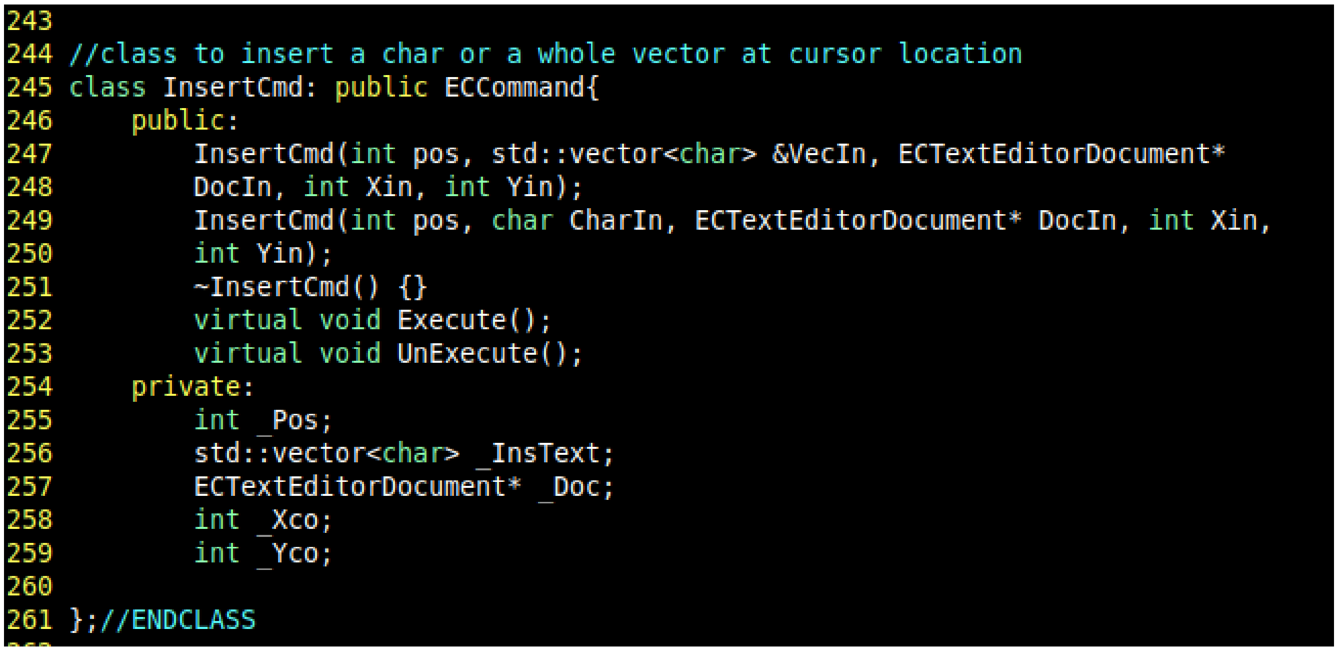
- Here you can see the Ticket class, it contains points to pretty much every major class within this text editor. This is done so that a handler is able to access any of these objects in order to send them commands.

Commands and The Command History: In order to implement undo/redo functionality, I built a Command History. This class keeps track of each command from the handlers of the previous section. Each of these commands are a sub-class of the Command class, meaning that they have a specified way they execute and Unexecute. When a command in undone, the Unexecute function of that command is called.



```
class ECCommand
{
public:
    virtual ~ECCommand() {}
    virtual void Execute() = 0;
    virtual void UnExecute() = 0;
};




class ECCommandHistory
{
public:
    ECCommandHistory();
    virtual ~ECCommandHistory();
    bool Undo();
    bool Redo();
    void ExecuteCmd( ECCommand *pCmd );

private:
    std::vector<ECCommand*>* _commandList;
    int _cPointer;
};

```

- Here are both the base command class and the Command History class.



```

//class to insert a char or a whole vector at cursor location
class InsertCmd: public ECCommand{
    public:
        InsertCmd(int pos, std::vector<char> &VecIn, ECTextEditorDocument*
        DocIn, int Xin, int Yin);
        InsertCmd(int pos, char CharIn, ECTextEditorDocument* DocIn, int Xin,
        int Yin);
        ~InsertCmd() {}
        virtual void Execute();
        virtual void UnExecute();
    private:
        int _Pos;
        std::vector<char> _InsText;
        ECTextEditorDocument* _Doc;
        int _Xco;
        int _Yco;

};//ENDCLASS
```

- Here you can see an example of a Command sub-class, there are multiple inputs when creating this command, bu the handler which creates them knows this and is able to provide. You can see that it has the inherited Execute and Unexecute functions which make it work within the Command History.