Our purpose in this project is to converting a context free grammar to its equivalent Chomsky normal form. We will use C++ language for this purpose.

First of all every grammar in Chomsky normal form is context-free and conversely, every context free grammar can be transformed into an equivalent one which is in Chomsky normal form. A context free grammar is in Chomsky Normal Form if all production rules satisfy one of the following conditions:

- A non-terminal generating a terminal
- A non-terminal generating two non-terminals
- Start symbol generating ε

In the beginning of our program we first must get an input file. For this we will use “fstream” library.

```
#include<fstream>
```

We create a string named text for the lines we will get by reading lines and we have a ifstream object named readFile to open file we want to read.

```
string text;
```

```
ifstream readFile("G1.txt");
```

In a while loop we assign the lines to text string.

```
while (getline (readFile, text)){
```

Now we can create a class of CFG (context free grammer) the attributes of this class will be as shown in the picture.

```
class CFG{
    public:
    vector <string> nonTerminals;
    vector <string> terminals;
    vector<string> rules;
    string start;

};
```

We are using vectors because Non-terminals, terminals and rules can be in different sizes therefore we will need a dynamic memory. But start attribute will always be one string so we can leave it as a string type.

After we create the CFG class now we can assign the file lines to the class attributes. For this purpose we can use the while loop we created before. In this while loop first we will set a state with the help of if/else conditions. If the value of the text variable is “NON-TERMINAL”, “TERMINAL”, “RULES” or “START” we will change the state variables value in order to set the correct values to correct attributes.

```
if(text=="NON-TERMINAL"){
    state="nt";
    continue;
}
else if(text=="TERMINAL"){
    state = "t";
    continue;
}
else if(text == "RULES"){
    state = "r";
    continue;
}
else if(text == "START"){
    state = "s";
    continue;
}
```

After deciding the set value we can use the continue method in order to not assign the headers to class attributes. After this we can push the values to the class vectors.

```
  if(state == "nt"){
    cfg.nonTerminals.push_back(text);
}
else if(state == "t"){
    cfg.terminals.push_back(text);
}
else if(state == "r"){
    cfg.rules.push_back(text);
}
else if(state =="s"){
    cfg.start = text;
}
```

Now we can create CNF class to store our converted CFG. For this we can simply create same attributes from CFG class.

```
class CNF{
    public:
    vector<string> nonTerminals;
    vector<string> terminals;
    vector<string> rules;
    string start;
```

After CNF class we will continue with adding start symbol and we will push it to non-terminal vector.

```
cnf.start = cfg.start+"0";
cnf.nonTerminals.push_back(cnf.start);
```

We can add other non-terminals from cfg object.

```
for(auto i = cfg.nonTerminals.begin(); i!= cfg.nonTerminals.end();i++){
    cnf.nonTerminals.push_back(*i);
}
```

And also the terminals because they won't change.

```
for(auto i = cfg.terminals.begin(); i!= cfg.terminals.end(); i++){
    cnf.terminals.push_back(*i);
}
```

Lastly we can push the first rule which is "S0:S"

```
cnf.rules.push_back(cnf.start+":"+cfg.start);
```

Now it's time to remove epsilon symbol from the production rules. For this we create a "removeEpsilon" function and we send "cfg" and "cnf" objects as parameters.

```
removeEpsilon(cfg, cnf);
```

In this function we first create a vector named "eNonTerms" for non-terminals which are includes epsilon symbol in their rules. If they have an epsilon we add their non-terminals to this vector.

Now we can search for epsilon symbol in the rules of context free grammar. We will use nested for loops for this process. One for changing rule lines in the vector and one for searching "e" char in the string.

```
int counter=0;

for(auto i = cfg.rules.begin(); i<cfg.rules.end(); i++){

    for(int j=0; j< (*i).length(); j++){
        if((*i)[j] == 'e'){
            eNonTerms.push_back((*i)[0]);

            cfg.rules.erase(cfg.rules.begin()+counter);

        }
    }
counter++;
}
```

When we find the “e” symbol we can pop the vector value and assign the non-terminal symbol to “eNonTerms” vector. After that we can push all rules of “cfg” to “cnf”.

```
for(auto i = cfg.rules.begin(); i != cfg.rules.end(); i++){
    cnf.rules.push_back(*i);
}
```

And we change the rules in order to not have non-terminal which includes epsilon.

```
for(auto i =eNonTerms.begin(); i<eNonTerms.end(); i++){
    for(auto j=cfg.rules.begin(); j<cfg.rules.end(); j++){
        for(int k=2; k< (*j).length(); k++){
            if((*j)[k] == *i){
                cnf.rules.push_back((*j).substr(0,k)+(*j).substr(k+1,(*j).length()));
            }
        }
    }
}
```

With substring method, we can organize rules by not having the non-terminal symbol.

And finally we remove units. For this we search for if any rule has single non-terminal at it's right hand side. If there is we copy the right hand side of the found non-terminal and push back of the vector with Non-terminal symbol that includes it.

```
//search for unit remove

for(auto i= cnf.rules.begin(); i<cnf.rules.end(); i++){
    if((*i).length() == 3){
        //search for non-terminal on the rhs

        for(auto j=cnf.rules.begin(); j<cnf.rules.end(); j++){
            if((*j)[0] == (*i)[2]){
                cnf.rules.push_back((*i)[0] + ":" + (*j).substr(2,(*j).length()));
            }
        }
    }
}
```