OS LAB 8

Submitted by : Ridhima Kohli
B19CSE071

The files for question 1 are by the same names as that of the algorithm

Fcfs - first come first serve policy

Sstf - shortest seek time : closest cylinder seeked first

Scan - once a direction is chosen it is continued until end is hit and then direction changes

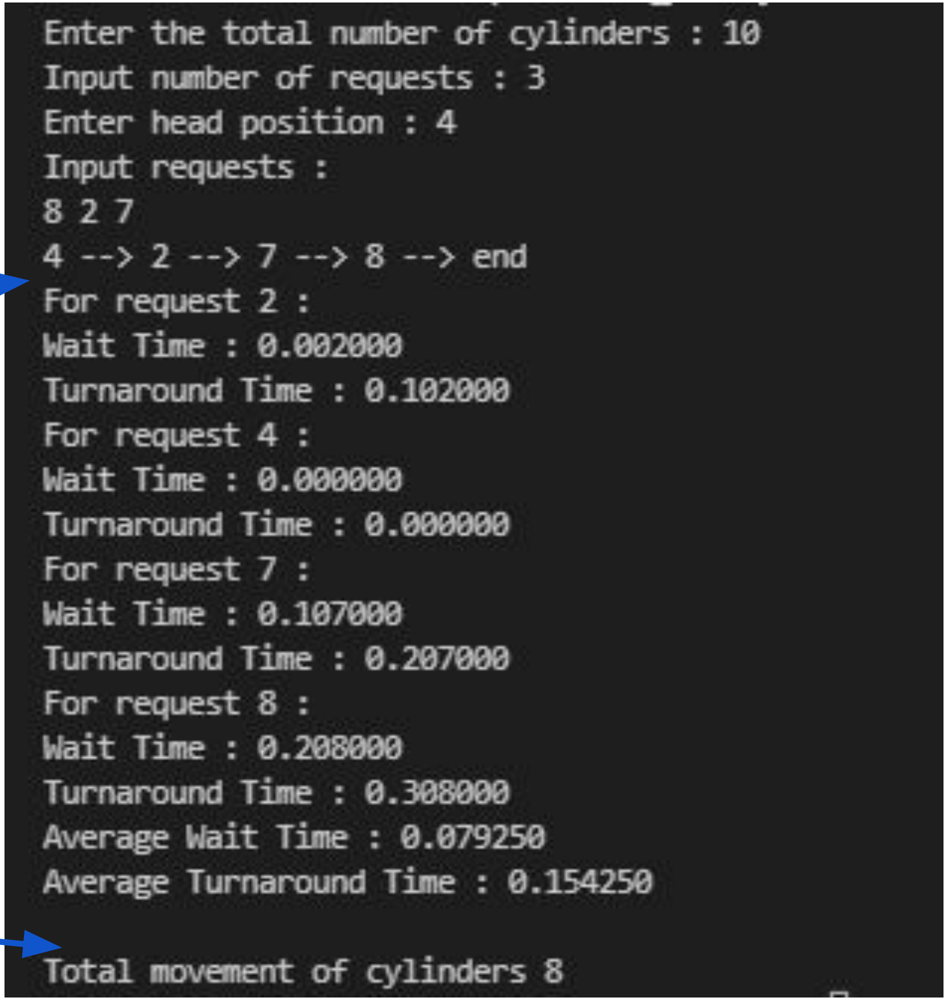
Cscan - once a direction is chosen it is continued , when end hits , the head jumps to other end and continues in same direction , thus making it circular scan

SCAN Disk Scheduling

Sequence of requests

Individual and average Waiting and turnaround time

Cylinder movement



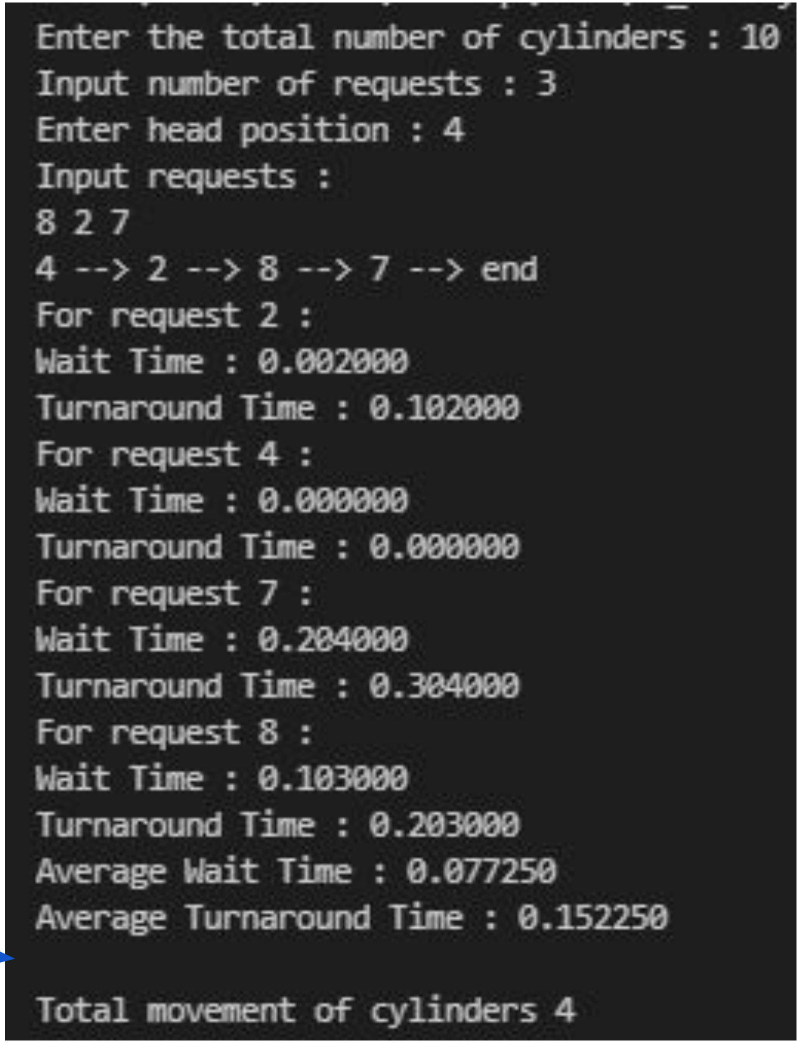
```
Enter the total number of cylinders : 10
Input number of requests : 3
Enter head position : 4
Input requests :
8 2 7
4 --> 2 --> 7 --> 8 --> end
For request 2 :
Wait Time : 0.002000
Turnaround Time : 0.102000
For request 4 :
Wait Time : 0.000000
Turnaround Time : 0.000000
For request 7 :
Wait Time : 0.107000
Turnaround Time : 0.207000
For request 8 :
Wait Time : 0.208000
Turnaround Time : 0.308000
Average Wait Time : 0.079250
Average Turnaround Time : 0.154250

Total movement of cylinders 8
```

C-SCAN Disk Scheduling

Sequence of requests

Individual and average Waiting and turnaround time

Cylinder movement



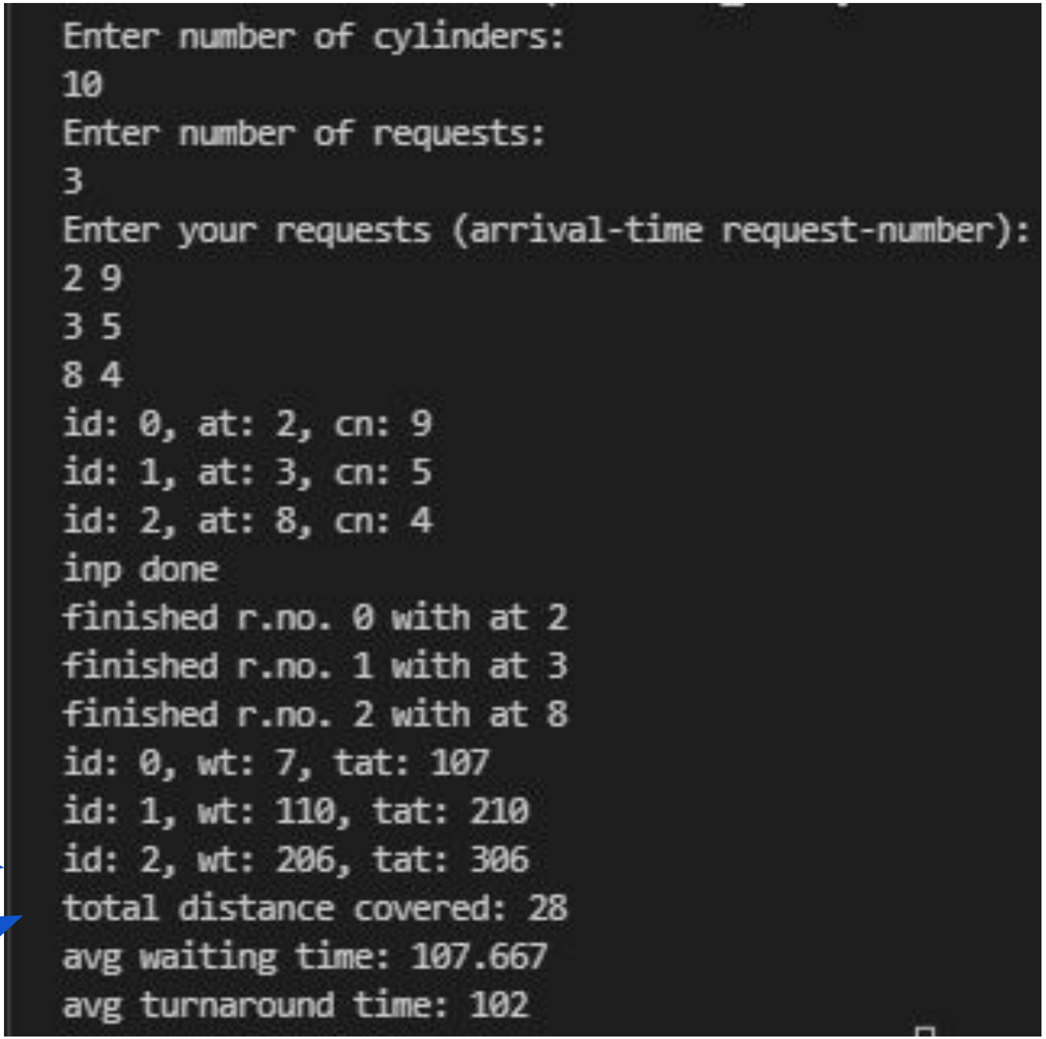
```
Enter the total number of cylinders : 10
Input number of requests : 3
Enter head position : 4
Input requests :
8 2 7
4 --> 2 --> 8 --> 7 --> end
For request 2 :
Wait Time : 0.002000
Turnaround Time : 0.102000
For request 4 :
Wait Time : 0.000000
Turnaround Time : 0.000000
For request 7 :
Wait Time : 0.204000
Turnaround Time : 0.304000
For request 8 :
Wait Time : 0.103000
Turnaround Time : 0.203000
Average Wait Time : 0.077250
Average Turnaround Time : 0.152250

Total movement of cylinders 4
```

SSTF
Disk Scheduling

Sequence of requests

Individual and average Waiting and turnaround time

Cylinder movement



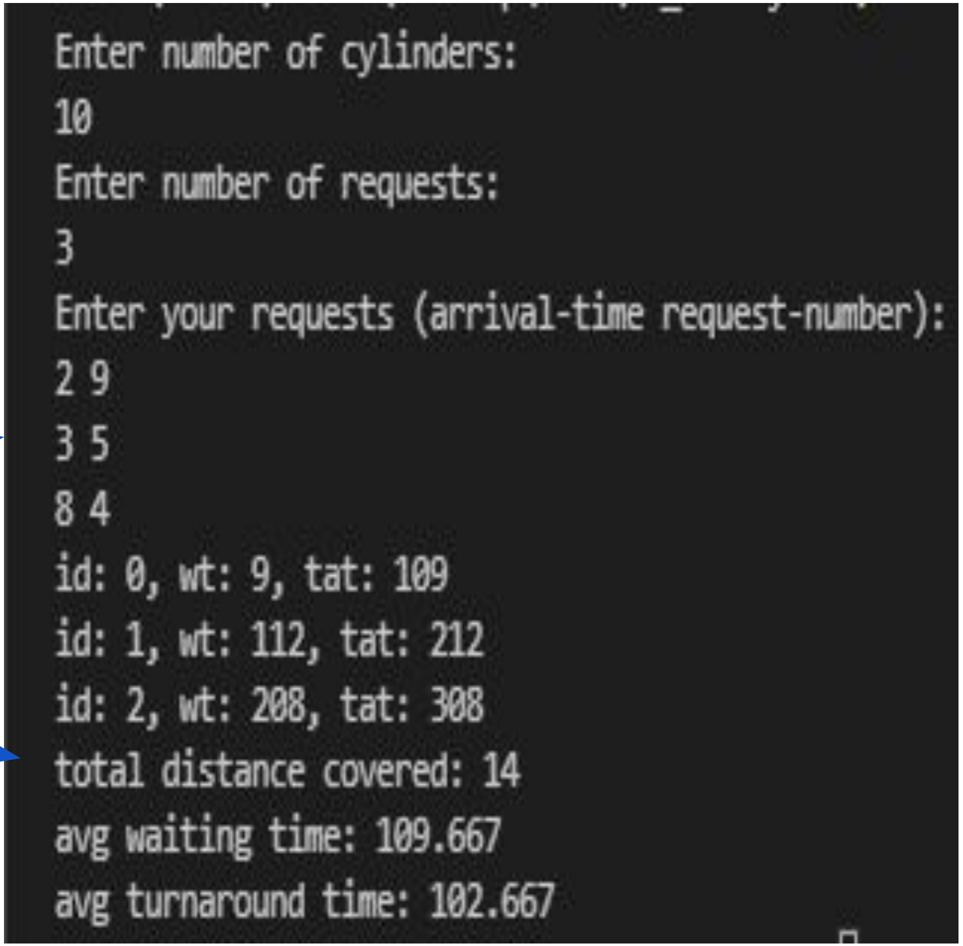
```
Enter number of cylinders:
10
Enter number of requests:
3
Enter your requests (arrival-time request-number):
2 9
3 5
8 4
id: 0, at: 2, cn: 9
id: 1, at: 3, cn: 5
id: 2, at: 8, cn: 4
inp done
finished r.no. 0 with at 2
finished r.no. 1 with at 3
finished r.no. 2 with at 8
id: 0, wt: 7, tat: 107
id: 1, wt: 110, tat: 210
id: 2, wt: 206, tat: 306
total distance covered: 28
avg waiting time: 107.667
avg turnaround time: 102
```

FCFS
Disk Scheduling

Sequence of requests

Individual and average Waiting and turnaround time

Cylinder movement

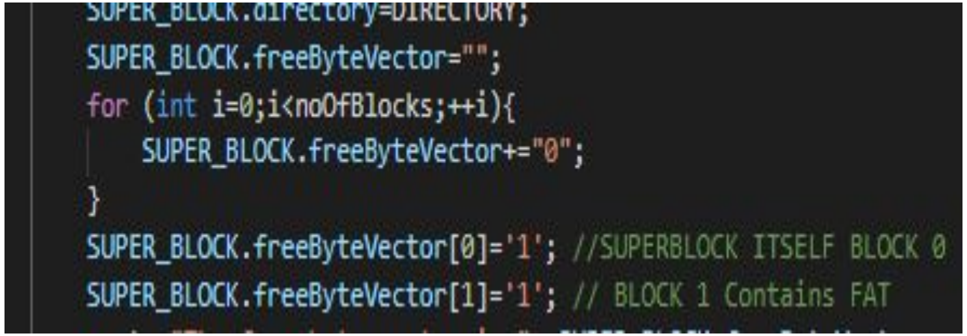


```
Enter number of cylinders:
10
Enter number of requests:
3
Enter your requests (arrival-time request-number):
2 9
3 5
8 4
id: 0, wt: 9, tat: 109
id: 1, wt: 112, tat: 212
id: 2, wt: 208, tat: 308
total distance covered: 14
avg waiting time: 109.667
avg turnaround time: 102.667
```

1. Representation of free blocks for FAT

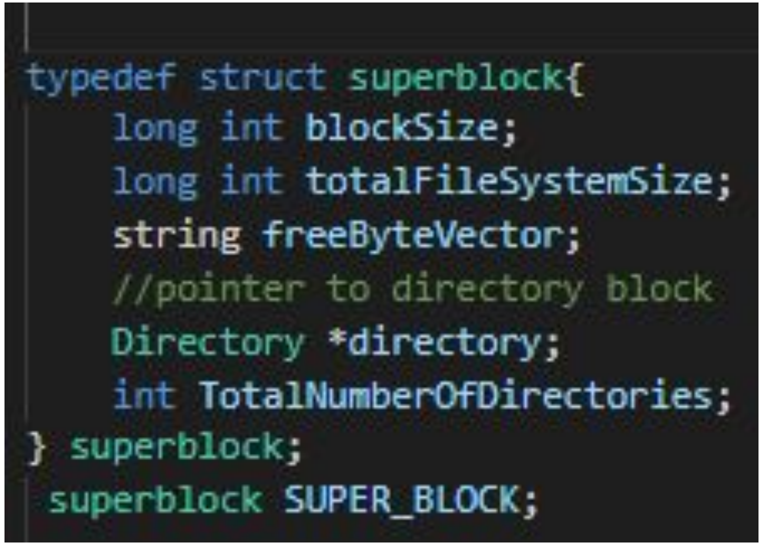
Ques 2 - file system

Part 1 : run file q2p1.cpp

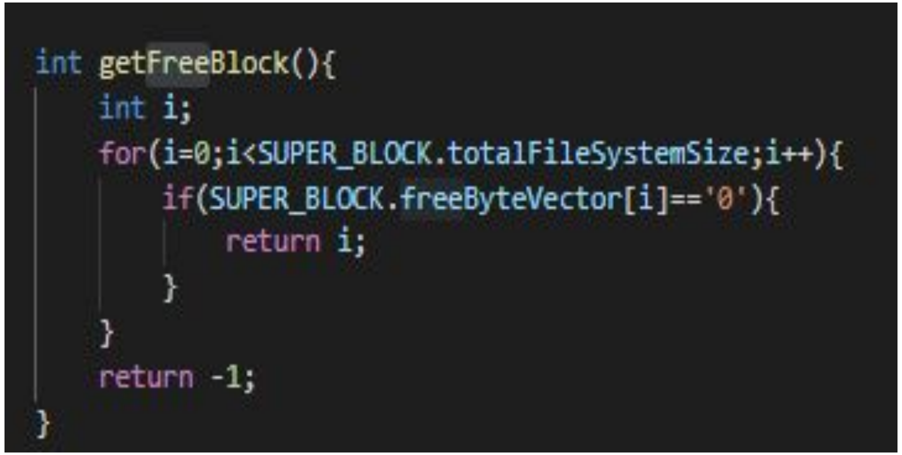


```
SUPER_BLOCK.directory=DIRECTORY;
SUPER_BLOCK.freeByteVector="";
for (int i=0;i<noOfBlocks;++i){
    SUPER_BLOCK.freeByteVector+="0";
}
SUPER_BLOCK.freeByteVector[0]='1'; //SUPERBLOCK ITSELF BLOCK 0
SUPER_BLOCK.freeByteVector[1]='1'; // BLOCK 1 Contains FAT
```

Declaration in super block as string
getFreeBlock function called to check free block for next assignment
Value initialization with 0 indicating free block



```
typedef struct superblock{
    long int blockSize;
    long int totalFileSystemSize;
    string freeByteVector;
    //pointer to directory block
    Directory *directory;
    int TotalNumberOfDirectories;
} superblock;
 superblock SUPER_BLOCK;
```

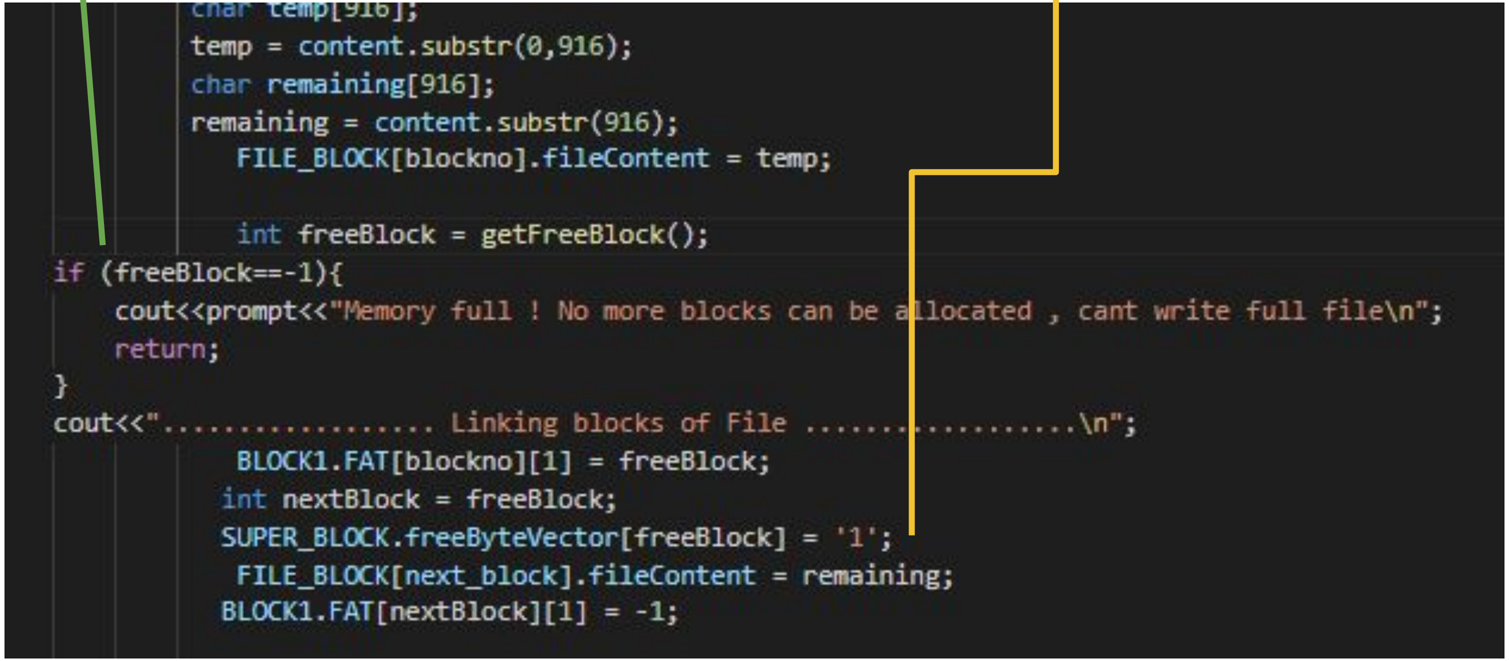


```
int getFreeBlock(){
    int i;
    for(i=0;i<SUPER_BLOCK.totalFileSystemSize;i++){
        if(SUPER_BLOCK.freeByteVector[i]=='0'){
            return i;
        }
    }
    return -1;
}
```

1. Representation of free blocks for FAT

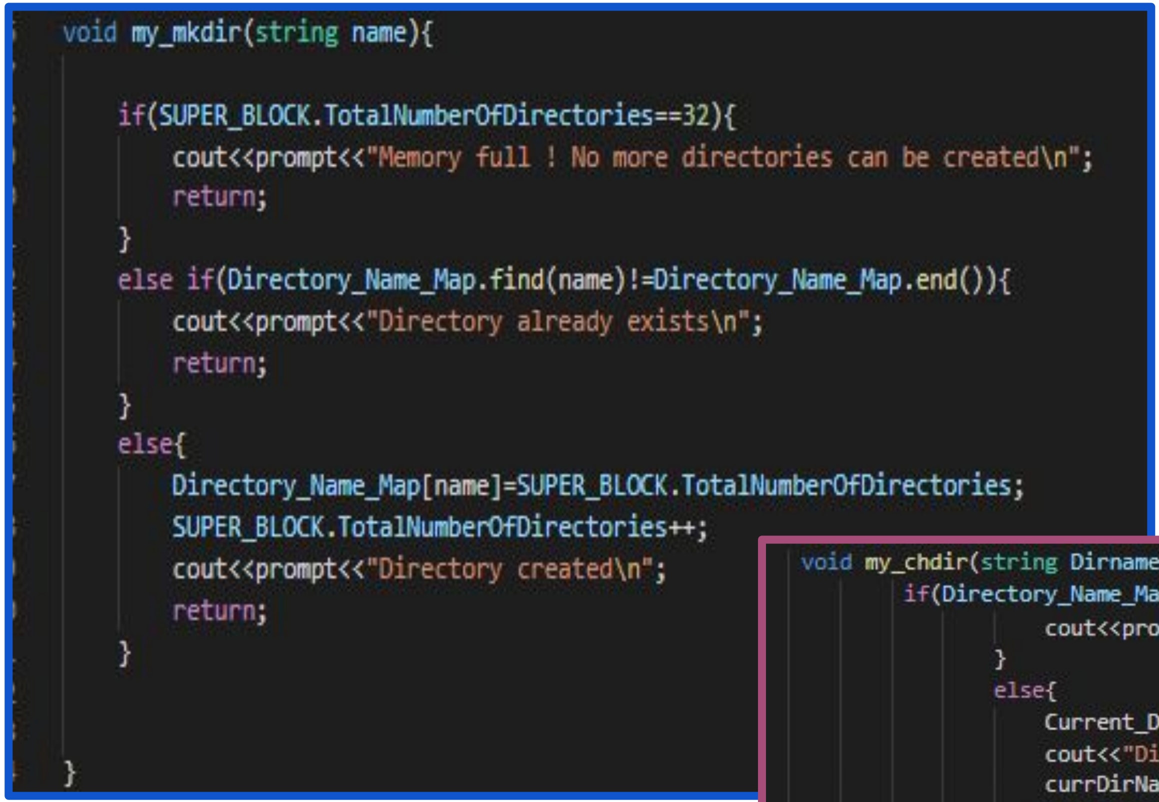
To check if all blocks are full

Free byte updated with new creation / deletion

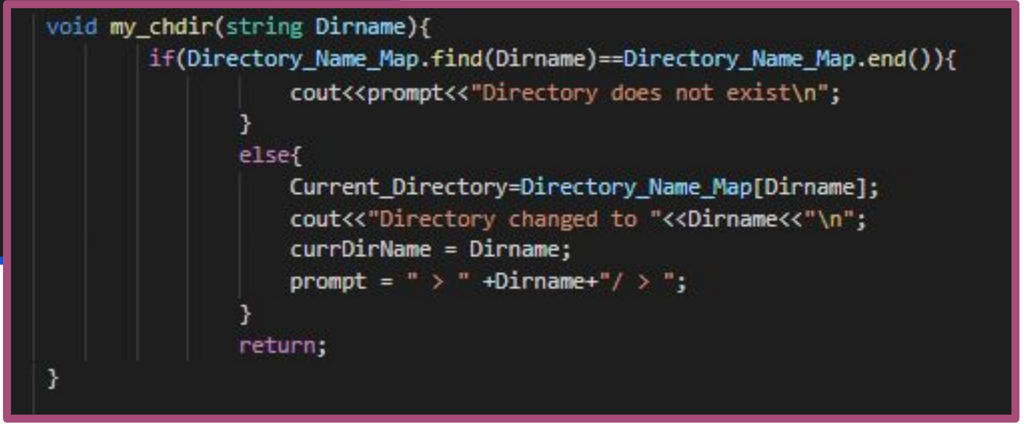


```
            char temp[916];
            temp = content.substr(0,916);
            char remaining[916];
            remaining = content.substr(916);
               FILE_BLOCK[blockno].fileContent = temp;

               int freeBlock = getFreeBlock();
if (freeBlock==-1){
    cout<<prompt<<"Memory full ! No more blocks can be allocated , cant write full file\n";
    return;
}
cout<<"................... Linking blocks of File ..................\n";
               BLOCK1.FAT[blockno][1] = freeBlock;
              int nextBlock = freeBlock;
              SUPER_BLOCK.freeByteVector[freeBlock] = '1';
               FILE_BLOCK[next_block].fileContent = remaining;
              BLOCK1.FAT[nextBlock][1] = -1;
```

2. APIs for FAT based allocation

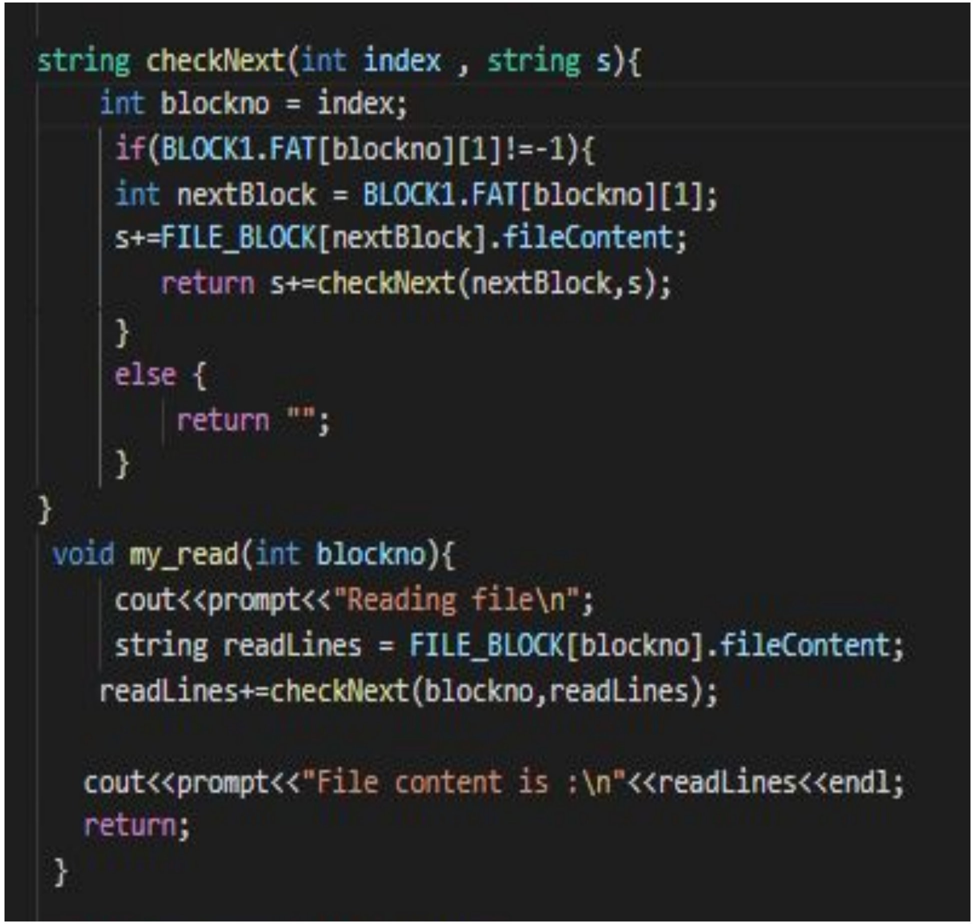


```
void my_mkdir(string name){

    if(SUPER_BLOCK.TotalNumberOfDirectories==32){
        cout<<prompt<<"Memory full ! No more directories can be created\n";
        return;
    }
    else if(Directory_Name_Map.find(name)!=Directory_Name_Map.end()){
        cout<<prompt<<"Directory already exists\n";
        return;
    }
    else{
        Directory_Name_Map[name]=SUPER_BLOCK.TotalNumberOfDirectories;
        SUPER_BLOCK.TotalNumberOfDirectories++;
        cout<<prompt<<"Directory created\n";
        return;
    }


}
```



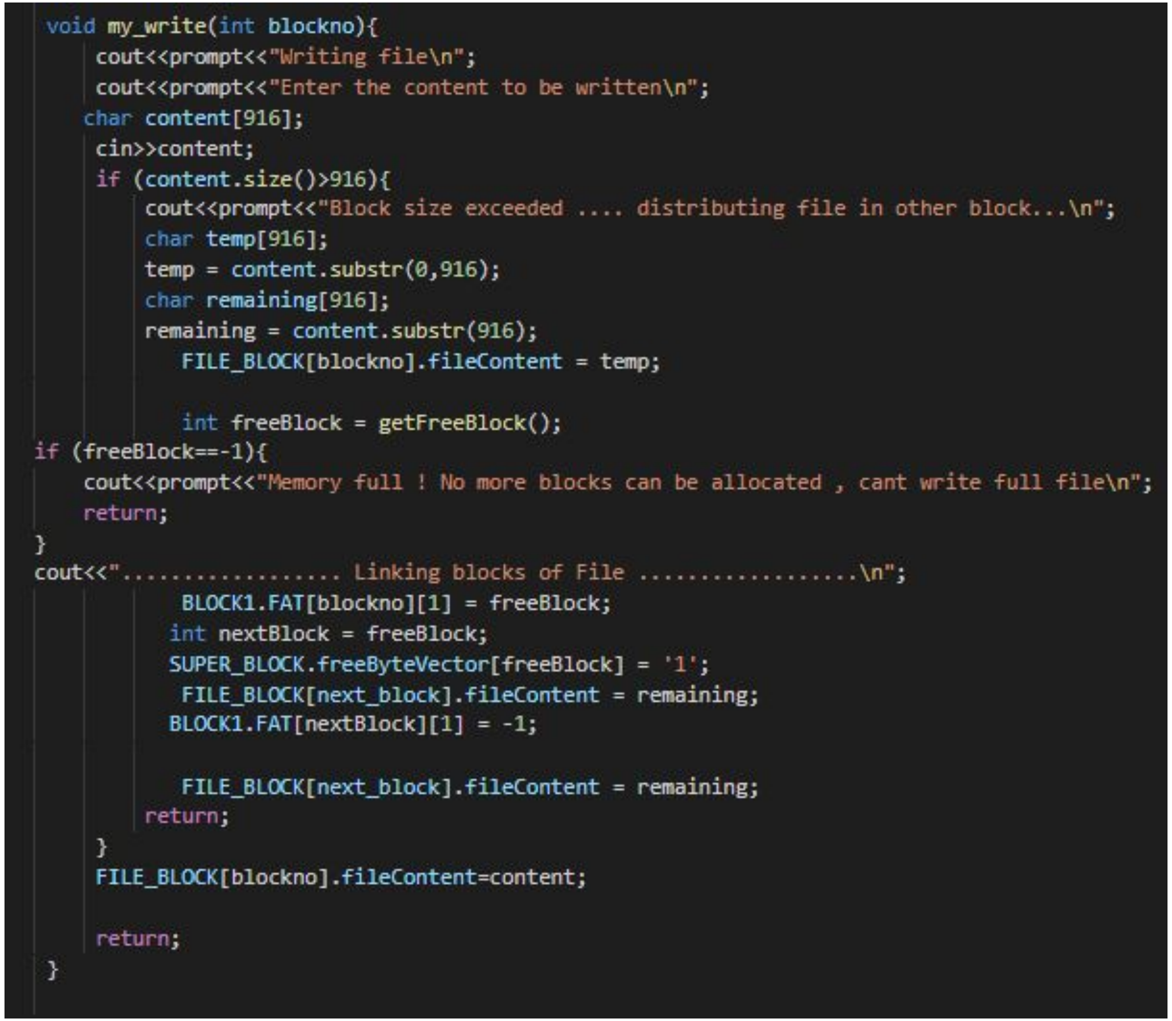
```
void my_chdir(string Dirname){
        if(Directory_Name_Map.find(Dirname)==Directory_Name_Map.end()){
                cout<<prompt<<"Directory does not exist\n";
            }
            else{
                Current_Directory=Directory_Name_Map[Dirname];
                cout<<"Directory changed to "<<Dirname<<"\n";
                currDirName = Dirname;
                prompt = " > " +Dirname+"/ > ";
            }
            return;
}
```

2. APIs for FAT based allocation



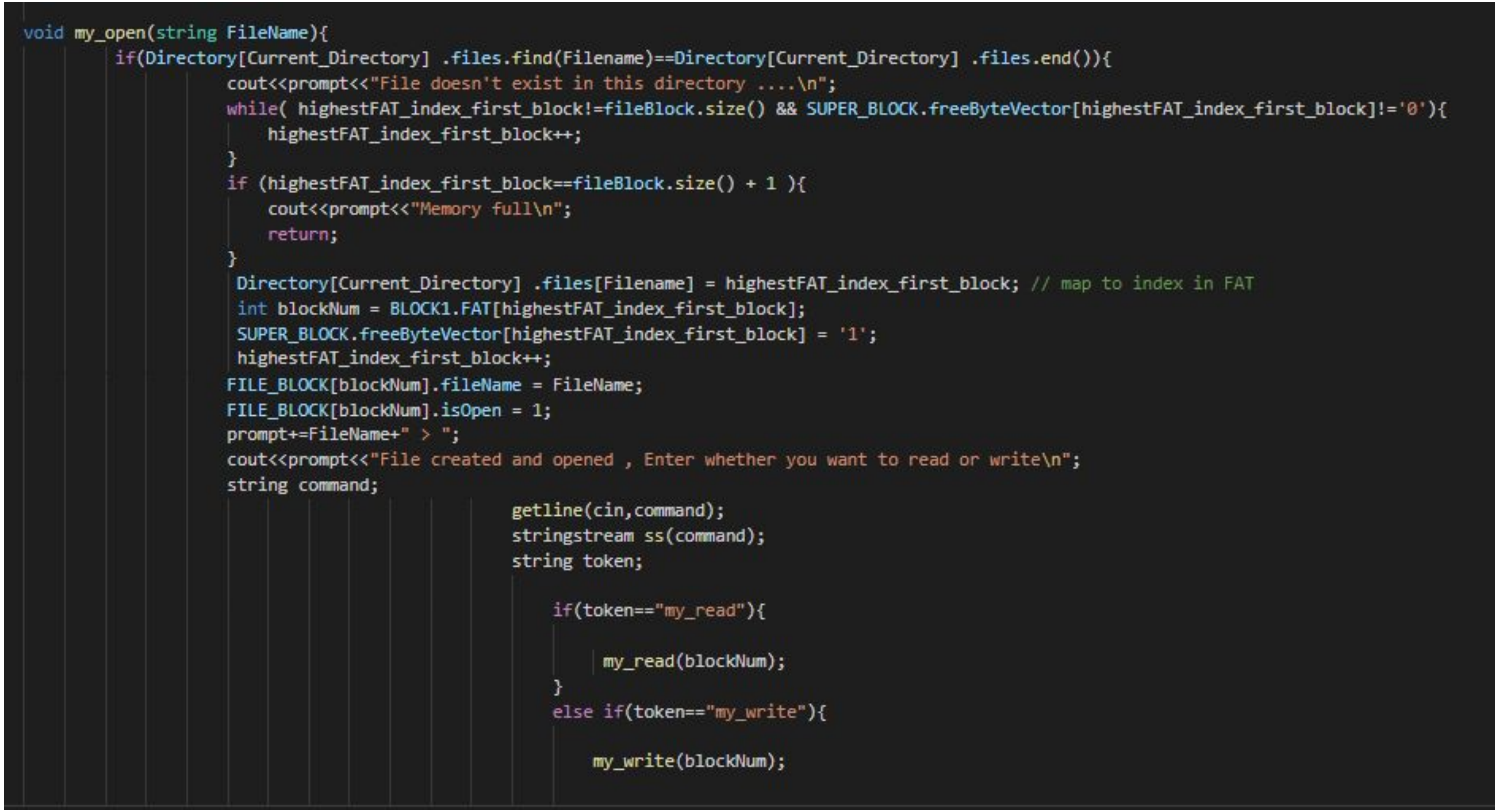
```
string checkNext(int index , string s){
    int blockno = index;
     if(BLOCK1.FAT[blockno][1]!=-1){
     int nextBlock = BLOCK1.FAT[blockno][1];
     s+=FILE_BLOCK[nextBlock].fileContent;
        return s+=checkNext(nextBlock,s);
     }
     else {
         return "";
     }
}
 void my_read(int blockno){
     cout<<prompt<<"Reading file\n";
     string readLines = FILE_BLOCK[blockno].fileContent;
    readLines+=checkNext(blockno,readLines);

   cout<<prompt<<"File content is :\n"<<readLines<<endl;
   return;
 }
```

2. APIs for FAT based allocation



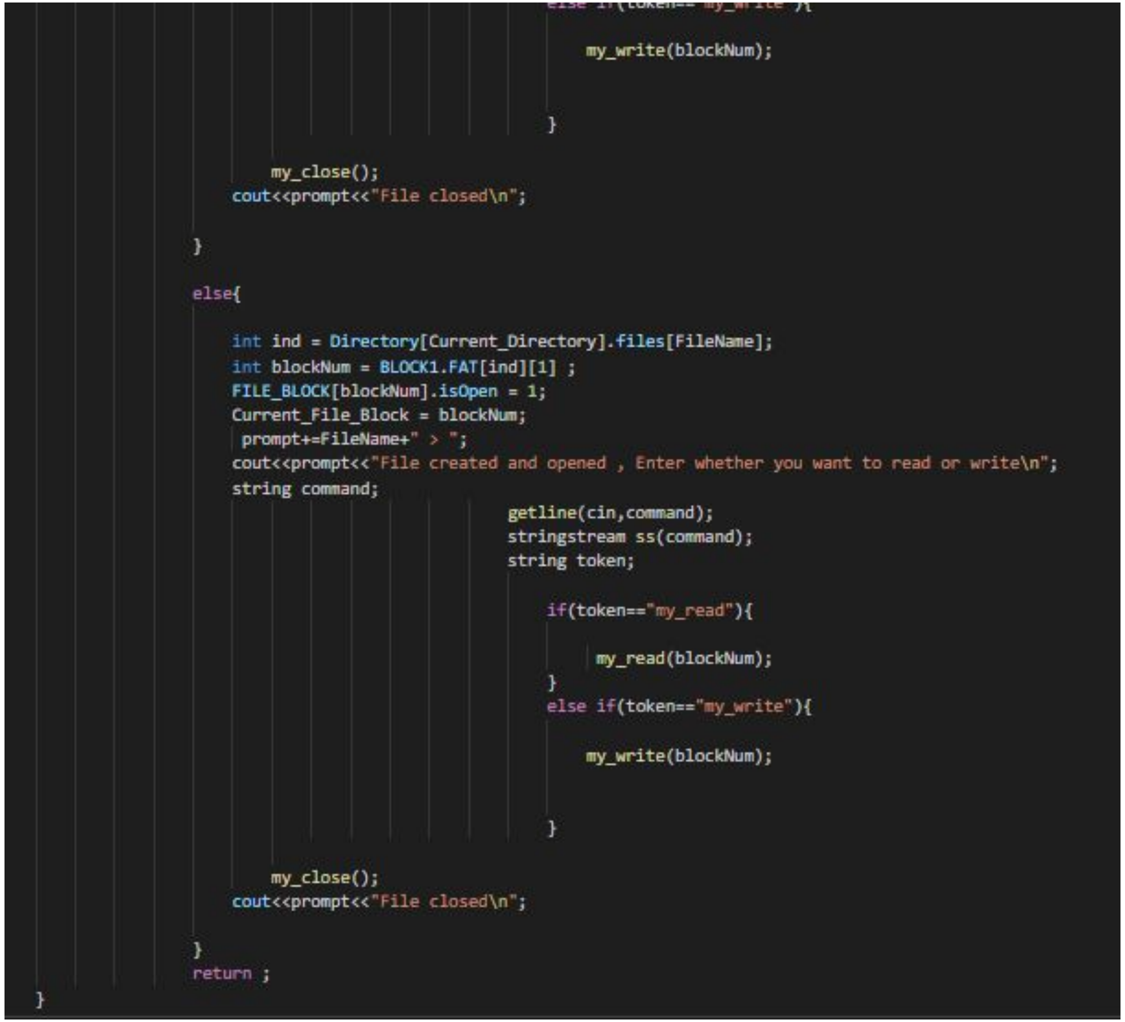
```
 void my_write(int blockno){
     cout<<prompt<<"Writing file\n";
     cout<<prompt<<"Enter the content to be written\n";
    char content[916];
     cin>>content;
     if (content.size()>916){
         cout<<prompt<<"Block size exceeded .... distributing file in other block...\n";
         char temp[916];
         temp = content.substr(0,916);
         char remaining[916];
         remaining = content.substr(916);
            FILE_BLOCK[blockno].fileContent = temp;

            int freeBlock = getFreeBlock();
if (freeBlock==-1){
    cout<<prompt<<"Memory full ! No more blocks can be allocated , cant write full file\n";
    return;
}
cout<<"................. Linking blocks of File .................\n";
            BLOCK1.FAT[blockno][1] = freeBlock;
           int nextBlock = freeBlock;
           SUPER_BLOCK.freeByteVector[freeBlock] = '1';
            FILE_BLOCK[next_block].fileContent = remaining;
           BLOCK1.FAT[nextBlock][1] = -1;

            FILE_BLOCK[next_block].fileContent = remaining;
         return;
     }
     FILE_BLOCK[blockno].fileContent=content;

     return;
 }
```

2. APIs for FAT based allocation



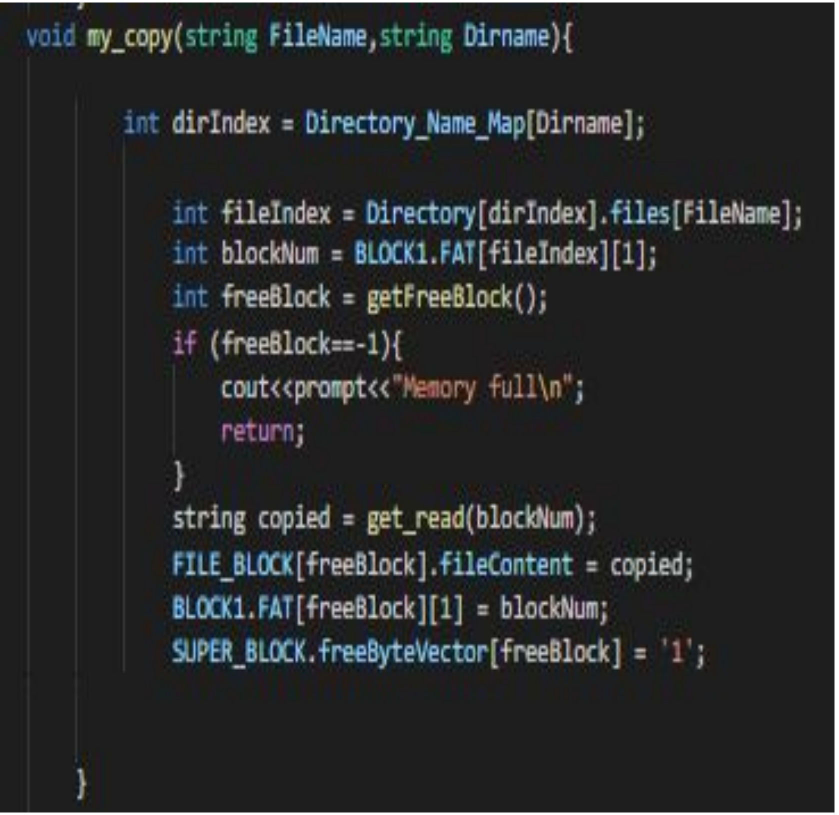
```
void my_open(string FileName){
        if(Directory[Current_Directory] .files.find(Filename)==Directory[Current_Directory] .files.end()){
                cout<<prompt<<"File doesn't exist in this directory ....\n";
                while( highestFAT_index_first_block!=fileBlock.size() && SUPER_BLOCK.freeByteVector[highestFAT_index_first_block]!='0'){
                    highestFAT_index_first_block++;
                }
                if (highestFAT_index_first_block==fileBlock.size() + 1 ){
                    cout<<prompt<<"Memory full\n";
                    return;
                }
                 Directory[Current_Directory] .files[Filename] = highestFAT_index_first_block; // map to index in FAT
                 int blockNum = BLOCK1.FAT[highestFAT_index_first_block];
                 SUPER_BLOCK.freeByteVector[highestFAT_index_first_block] = '1';
                 highestFAT_index_first_block++;
                FILE_BLOCK[blockNum].fileName = FileName;
                FILE_BLOCK[blockNum].isOpen = 1;
                prompt+=FileName+" > ";
                cout<<prompt<<"File created and opened , Enter whether you want to read or write\n";
                string command;
                                            getline(cin,command);
                                            stringstream ss(command);
                                            string token;

                                                if(token=="my_read"){

                                                    my_read(blockNum);
                                                }
                                                else if(token=="my_write"){

                                                   my_write(blockNum);
```

Previous continued

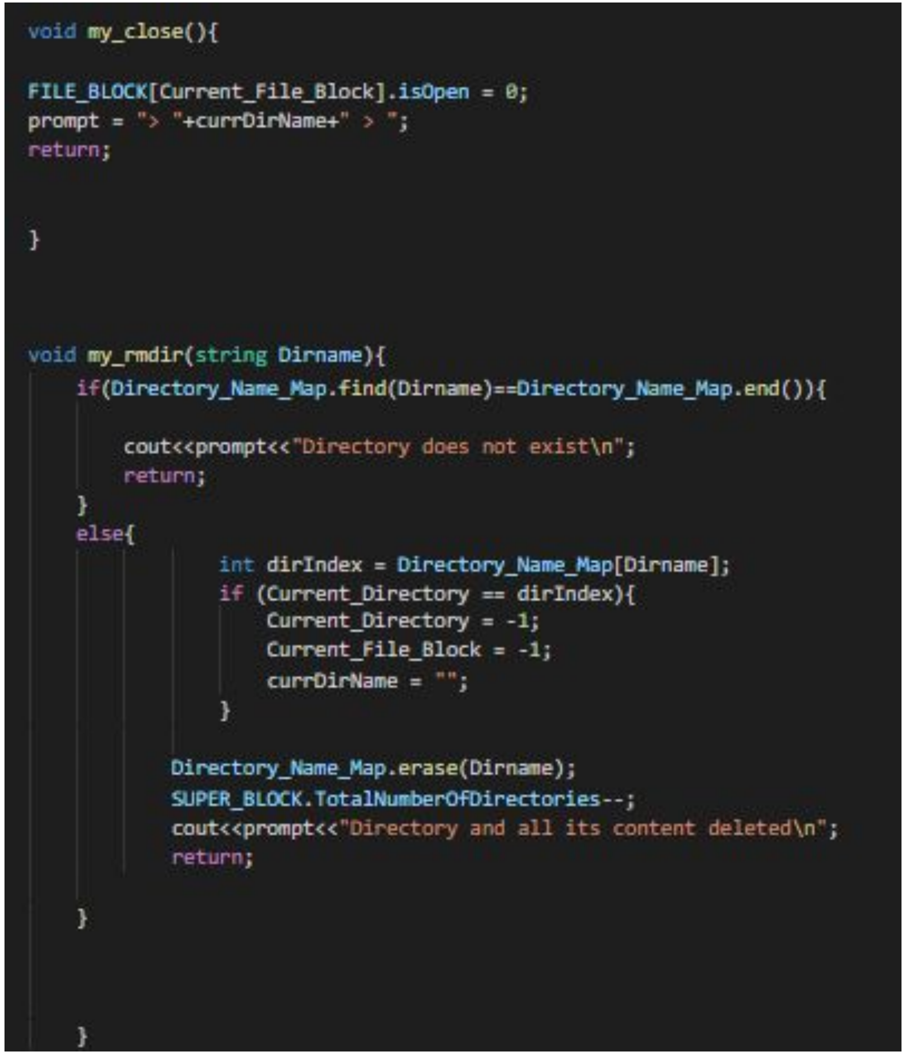


```
                            else if(token=="my_write"){

                                my_write(blockNum);


                            }

            my_close();
        cout<<prompt<<"File closed\n";

    }

    else{

        int ind = Directory[Current_Directory].files[FileName];
        int blockNum = BLOCK1.FAT[ind][1] ;
        FILE_BLOCK[blockNum].isOpen = 1;
        Current_File_Block = blockNum;
         prompt+=FileName+" > ";
        cout<<prompt<<"File created and opened , Enter whether you want to read or write\n";
        string command;
                        getline(cin,command);
                        stringstream ss(command);
                        string token;

                            if(token=="my_read"){

                                 my_read(blockNum);
                            }
                            else if(token=="my_write"){

                                my_write(blockNum);


                            }

            my_close();
        cout<<prompt<<"File closed\n";

    }
    return ;
}
```

2. APIs for FAT based allocation

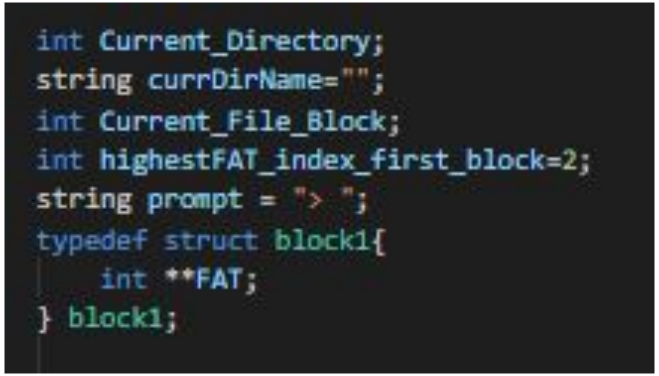


```
void my_copy(string FileName,string Dirname){

        int dirIndex = Directory_Name_Map[Dirname];

            int fileIndex = Directory[dirIndex].files[FileName];
            int blockNum = BLOCK1.FAT[fileIndex][1];
            int freeBlock = getFreeBlock();
            if (freeBlock==-1){
                cout<<prompt<<"Memory full\n";
                return;
            }
            string copied = get_read(blockNum);
            FILE_BLOCK[freeBlock].fileContent = copied;
            BLOCK1.FAT[freeBlock][1] = blockNum;
            SUPER_BLOCK.freeByteVector[freeBlock] = '1';


    }
```

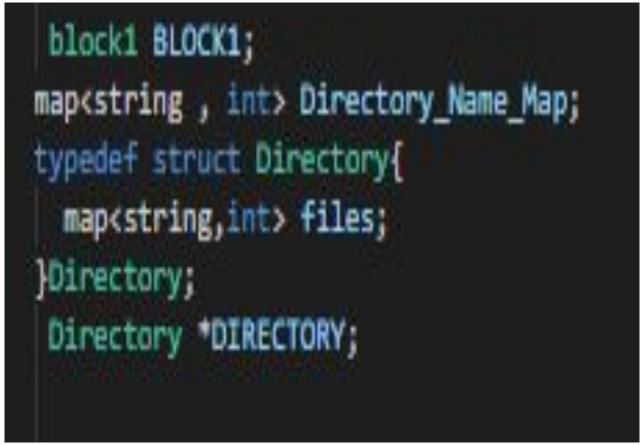


```
void my_close(){

FILE_BLOCK[Current_File_Block].isOpen = 0;
prompt = "> "+currDirName+" > ";
return;


}



void my_rmdir(string Dirname){
    if(Directory_Name_Map.find(Dirname)==Directory_Name_Map.end()){

        cout<<prompt<<"Directory does not exist\n";
        return;
    }
    else{
                int dirIndex = Directory_Name_Map[Dirname];
                if (Current_Directory == dirIndex){
                    Current_Directory = -1;
                    Current_File_Block = -1;
                    currDirName = "";
                }

            Directory_Name_Map.erase(Dirname);
            SUPER_BLOCK.TotalNumberOfDirectories--;
            cout<<prompt<<"Directory and all its content deleted\n";
            return;

    }



    }
```

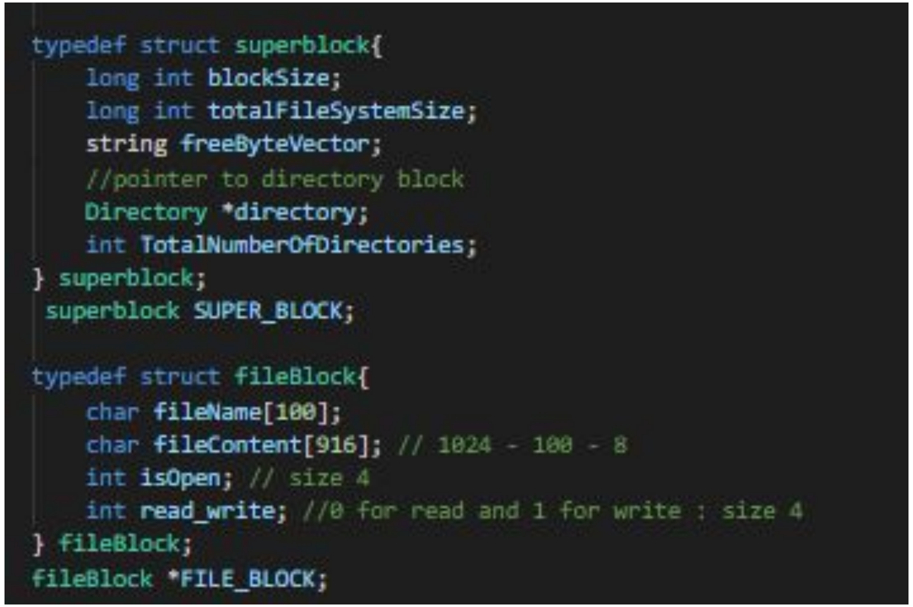
3. Overall implementation of FAT allocation



```
int Current_Directory;
string currDirName="";
int Current_File_Block;
int highestFAT_index_first_block=2;
string prompt = "> ";
typedef struct block1{
    int **FAT;
} block1;
```



```
block1 BLOCK1;
map<string , int> Directory_Name_Map;
typedef struct Directory{
  map<string,int> files;
}Directory;
 Directory *DIRECTORY;
```



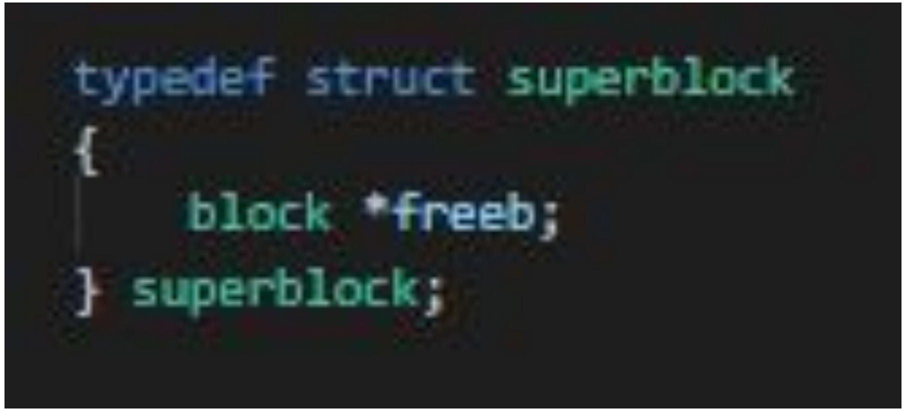
```
typedef struct superblock{
    long int blockSize;
    long int totalFileSystemSize;
    string freeByteVector;
    //pointer to directory block
    Directory *directory;
    int TotalNumberOfDirectories;
} superblock;
 superblock SUPER_BLOCK;

typedef struct fileBlock{
    char fileName[100];
    char fileContent[916]; // 1024 - 100 - 8
    int isOpen; // size 4
    int read_write; //0 for read and 1 for write : size 4
} fileBlock;
fileBlock *FILE_BLOCK;
```

Structs used for implementation
The directory pointer points to directory block
FAT is contained in block 1 while block 0 is super block

Ques 2 - file system : Part 2

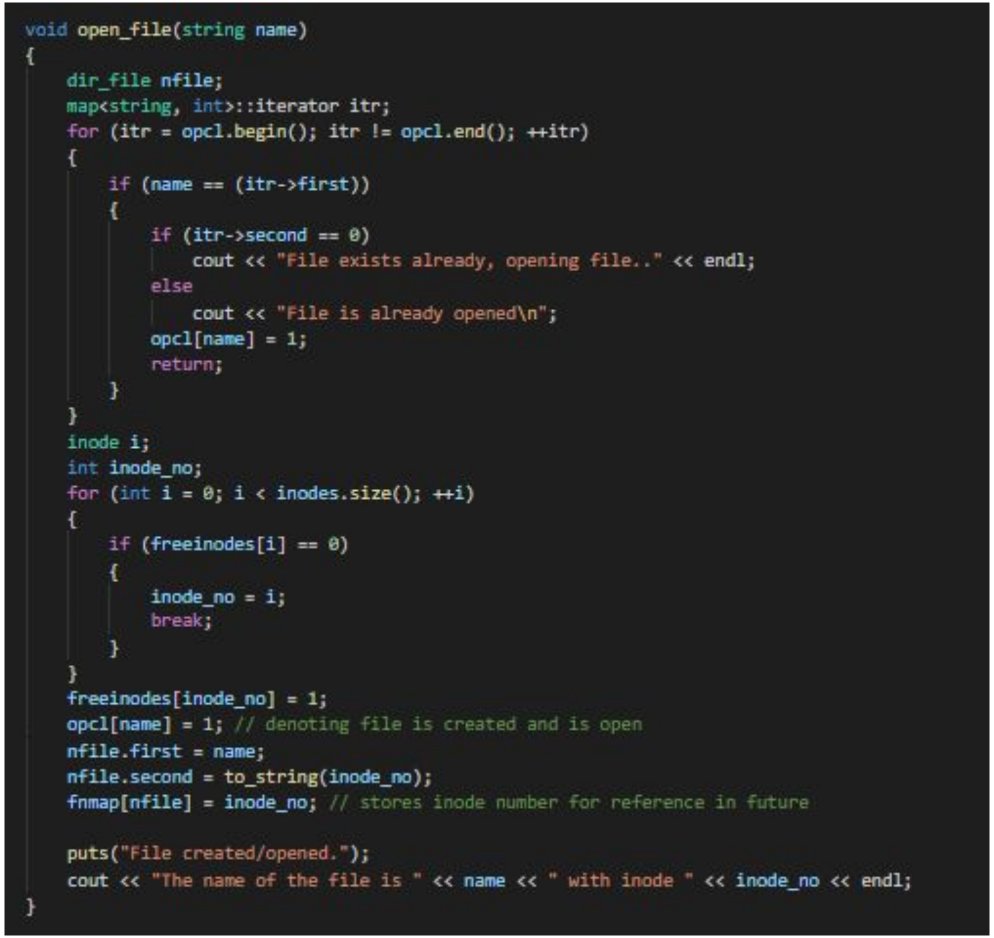
Run file q2p2.cpp

4. Representation of free blocks for i-node

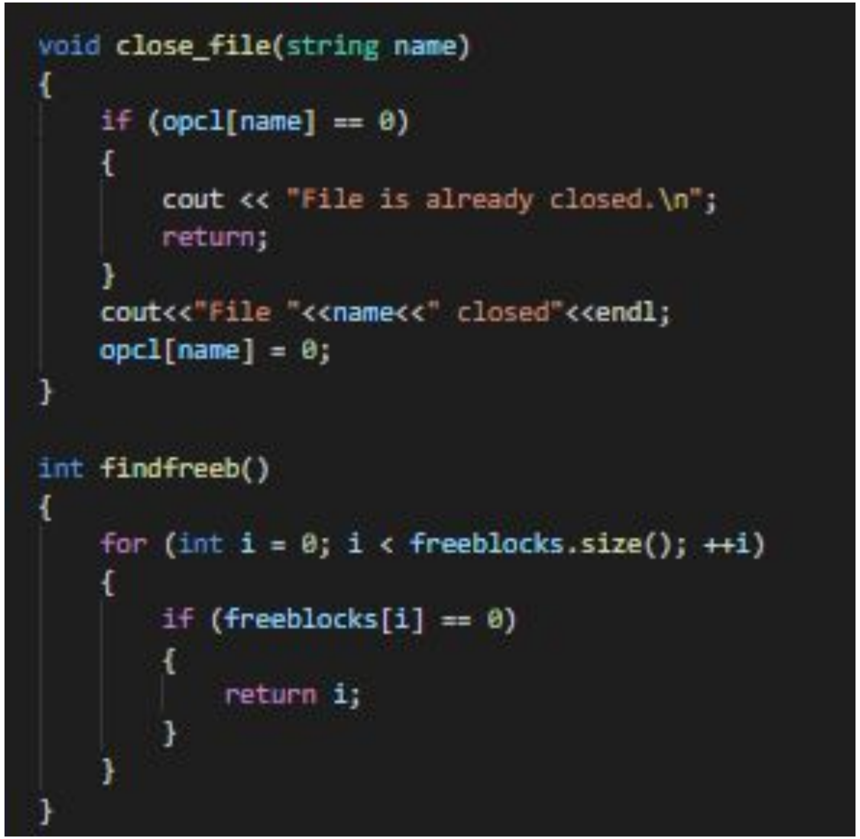


```
typedef struct superblock
{
    block *freeb;
} superblock;
```

5. APIs for i-node based allocation

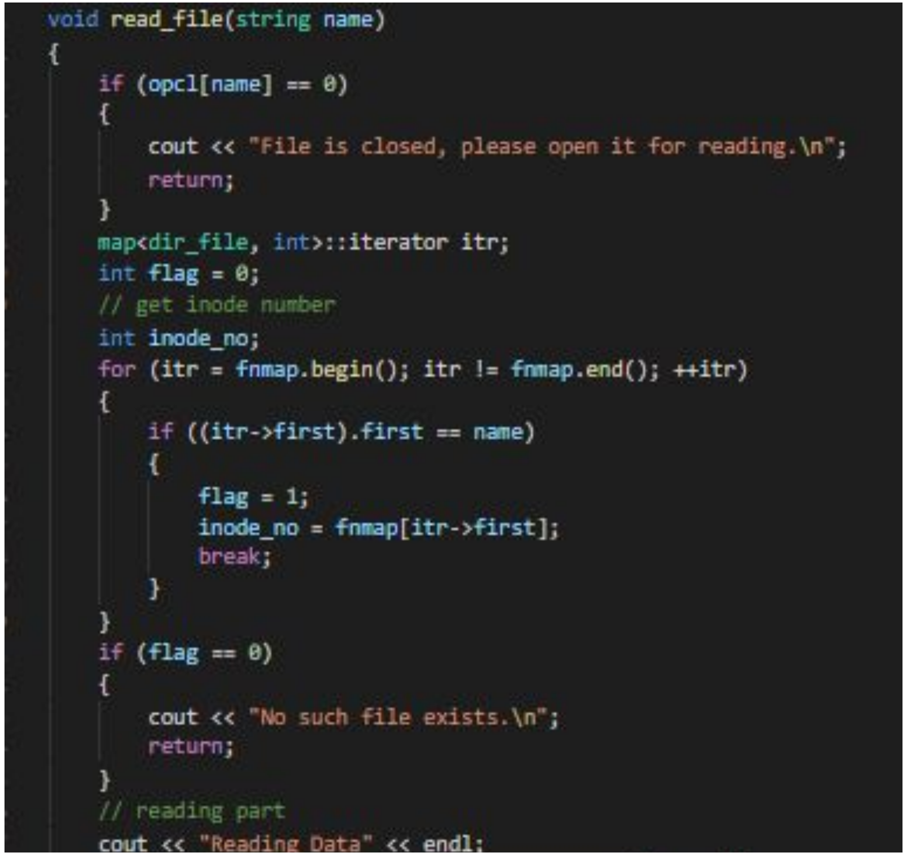


```cpp
void open_file(string name)
{
    dir_file nfile;
    map<string, int>::iterator itr;
    for (itr = opcl.begin(); itr != opcl.end(); ++itr)
    {
        if (name == (itr->first))
        {
            if (itr->second == 0)
                cout << "File exists already, opening file.." << endl;
            else
                cout << "File is already opened\n";
            opcl[name] = 1;
            return;
        }
    }
    inode i;
    int inode_no;
    for (int i = 0; i < inodes.size(); ++i)
    {
        if (freeinodes[i] == 0)
        {
            inode_no = i;
            break;
        }
    }
    freeinodes[inode_no] = 1;
    opcl[name] = 1; // denoting file is created and is open
    nfile.first = name;
    nfile.second = to_string(inode_no);
    fnmap[nfile] = inode_no; // stores inode number for reference in future

    puts("File created/opened.");
    cout << "The name of the file is " << name << " with inode " << inode_no << endl;
}
```

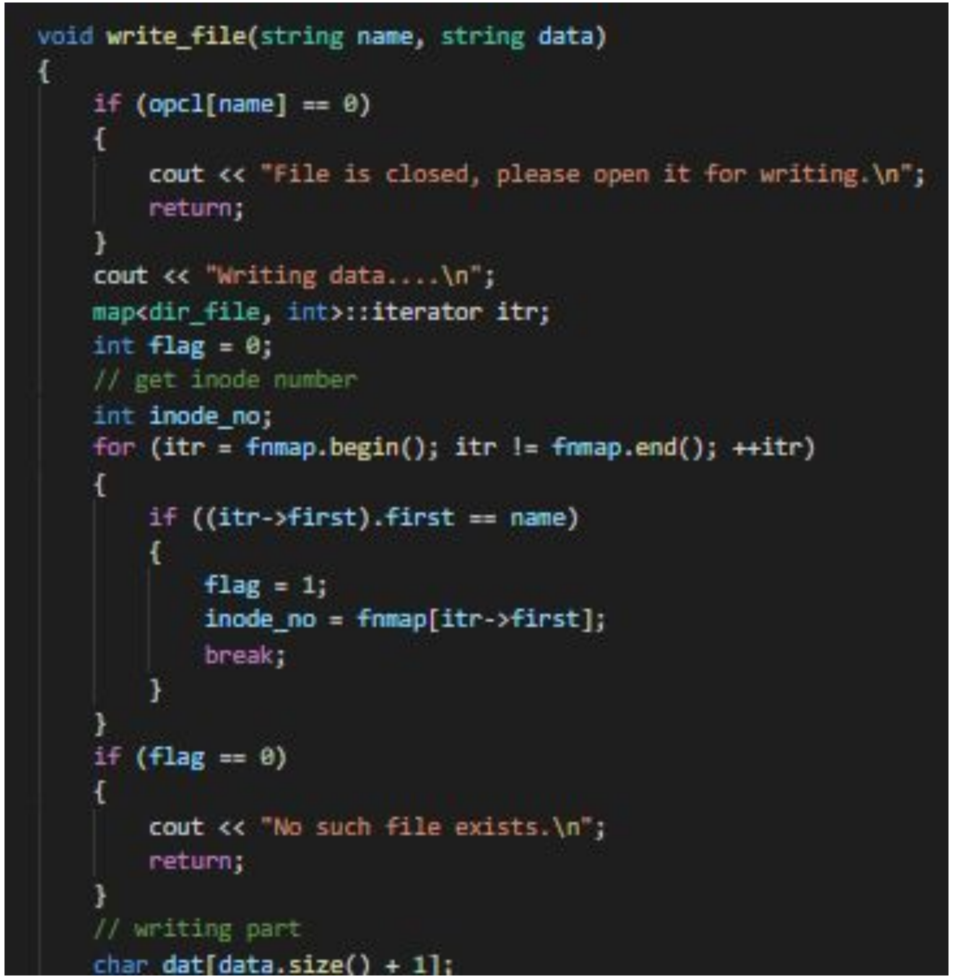


```cpp
void close_file(string name)
{
    if (opcl[name] == 0)
    {
        cout << "File is already closed.\n";
        return;
    }
    cout<<"File "<<name<<" closed"<<endl;
    opcl[name] = 0;
}

int findfreeb()
{
    for (int i = 0; i < freeblocks.size(); ++i)
    {
        if (freeblocks[i] == 0)
        {
            return i;
        }
    }
}
```

5. APIs for i-node based allocation

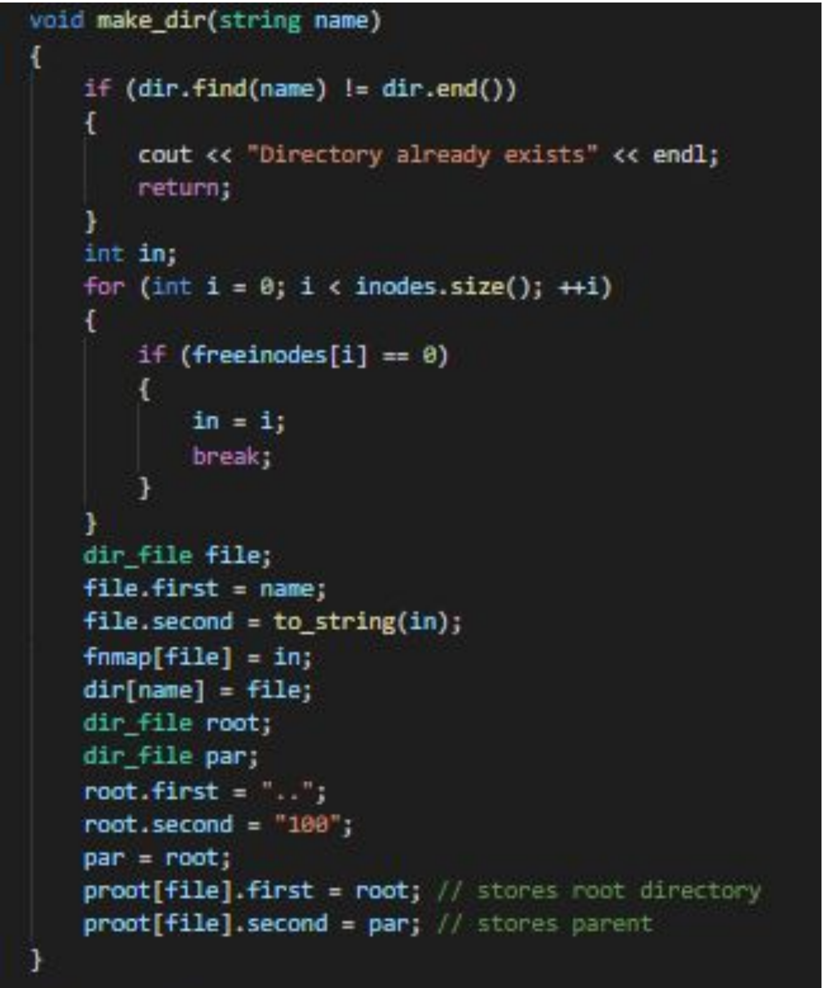


```
void read_file(string name)
{
    if (opcl[name] == 0)
    {
        cout << "File is closed, please open it for reading.\n";
        return;
    }
    map<dir_file, int>::iterator itr;
    int flag = 0;
    // get inode number
    int inode_no;
    for (itr = fnmap.begin(); itr != fnmap.end(); ++itr)
    {
        if ((itr->first).first == name)
        {
            flag = 1;
            inode_no = fnmap[itr->first];
            break;
        }
    }
    if (flag == 0)
    {
        cout << "No such file exists.\n";
        return;
    }
    // reading part
    cout << "Reading Data" << endl;
```

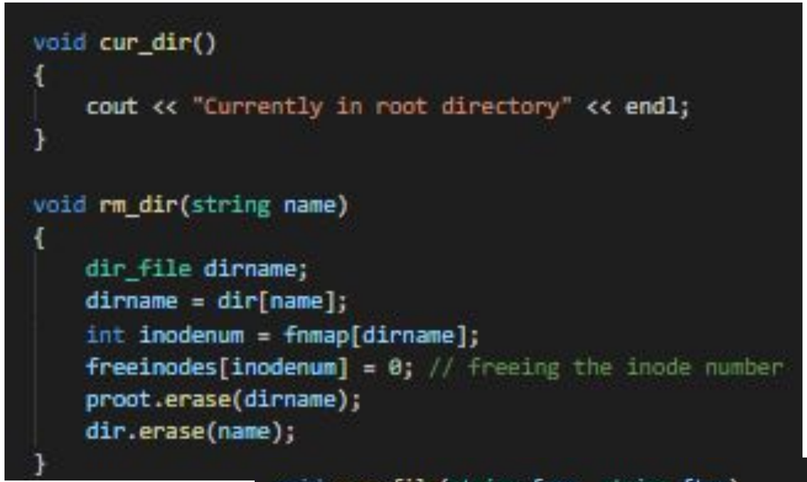


```
void write_file(string name, string data)
{
    if (opcl[name] == 0)
    {
        cout << "File is closed, please open it for writing.\n";
        return;
    }
    cout << "Writing data....\n";
    map<dir_file, int>::iterator itr;
    int flag = 0;
    // get inode number
    int inode_no;
    for (itr = fnmap.begin(); itr != fnmap.end(); ++itr)
    {
        if ((itr->first).first == name)
        {
            flag = 1;
            inode_no = fnmap[itr->first];
            break;
        }
    }
    if (flag == 0)
    {
        cout << "No such file exists.\n";
        return;
    }
    // writing part
    char dat[data.size() + 1];
```

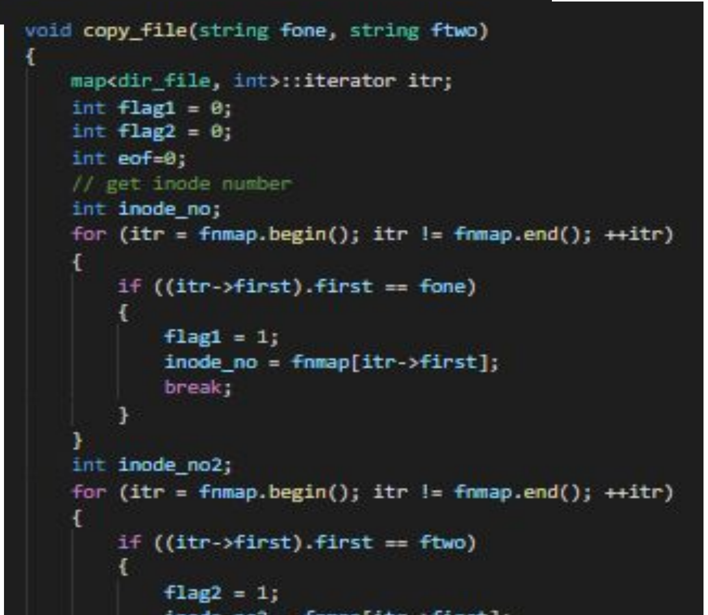
5. APIs for i-node based allocation



```
void make_dir(string name)
{
    if (dir.find(name) != dir.end())
    {
        cout << "Directory already exists" << endl;
        return;
    }
    int in;
    for (int i = 0; i < inodes.size(); ++i)
    {
        if (freeinodes[i] == 0)
        {
            in = i;
            break;
        }
    }
    dir_file file;
    file.first = name;
    file.second = to_string(in);
    fnmap[file] = in;
    dir[name] = file;
    dir_file root;
    dir_file par;
    root.first = "..";
    root.second = "100";
    par = root;
    proot[file].first = root; // stores root directory
    proot[file].second = par; // stores parent
}
```

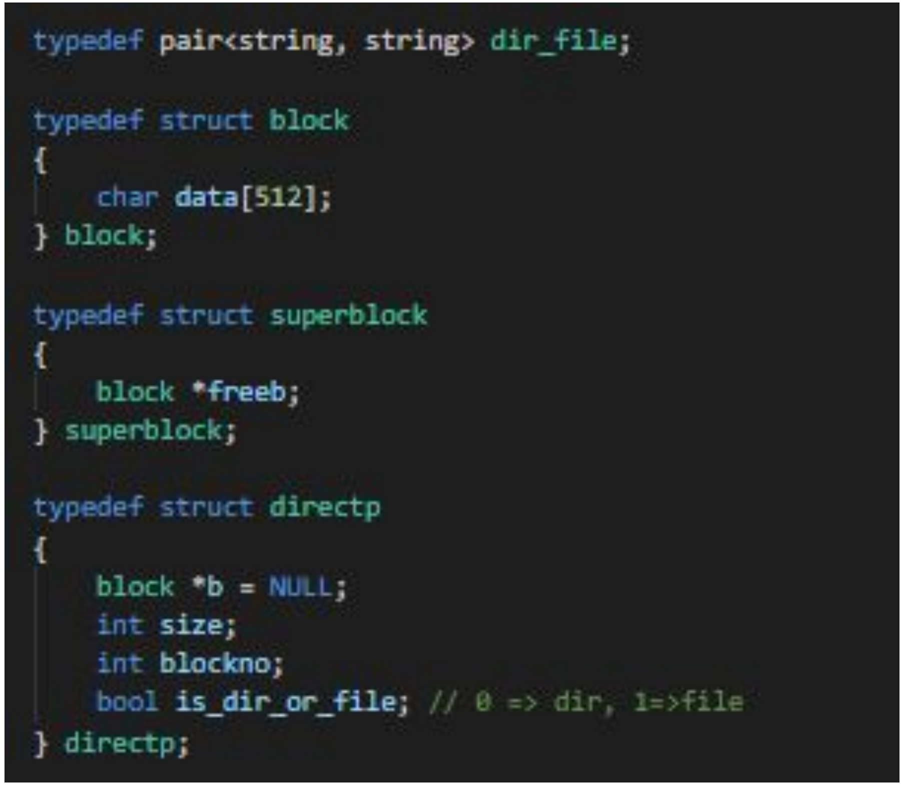


```
void cur_dir()
{
    cout << "Currently in root directory" << endl;
}

void rm_dir(string name)
{
    dir_file dirname;
    dirname = dir[name];
    int inodenum = fnmap[dirname];
    freeinodes[inodenum] = 0; // freeing the inode number
    proot.erase(dirname);
    dir.erase(name);
}
```

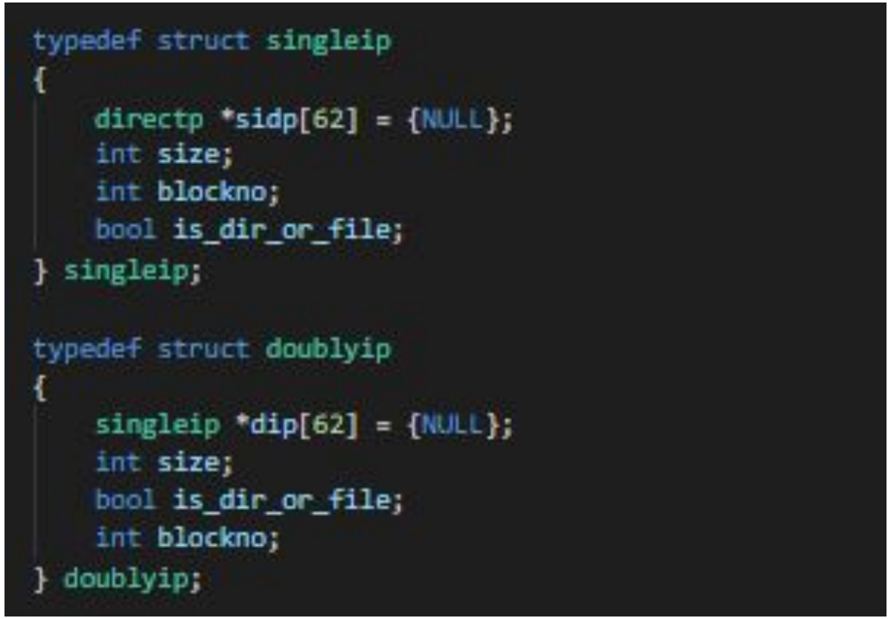


```
void copy_file(string fone, string ftwo)
{
    map<dir_file, int>::iterator itr;
    int flag1 = 0;
    int flag2 = 0;
    int eof=0;
    // get inode number
    int inode_no;
    for (itr = fnmap.begin(); itr != fnmap.end(); ++itr)
    {
        if ((itr->first).first == fone)
        {
            flag1 = 1;
            inode_no = fnmap[itr->first];
            break;
        }
    }
    int inode_no2;
    for (itr = fnmap.begin(); itr != fnmap.end(); ++itr)
    {
        if ((itr->first).first == ftwo)
        {
            flag2 = 1;
```

6. Overall implementation of i-node based allocation

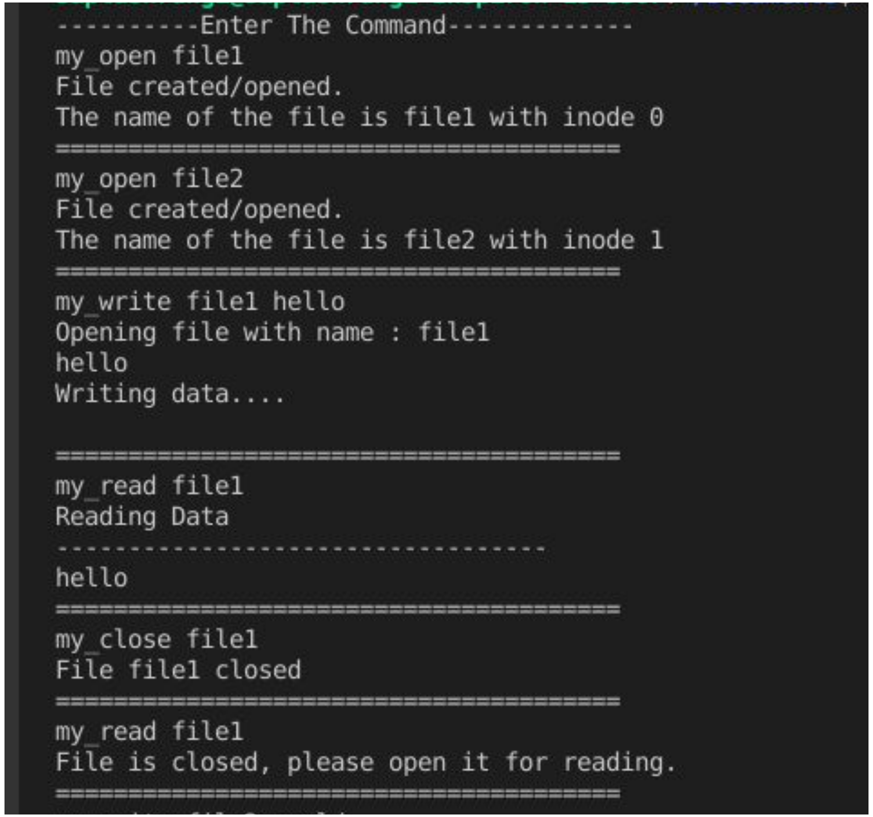


```
typedef pair<string, string> dir_file;

typedef struct block
{
    char data[512];
} block;

typedef struct superblock
{
    block *freeb;
} superblock;

typedef struct directp
{
    block *b = NULL;
    int size;
    int blockno;
    bool is_dir_or_file; // 0 => dir, 1=>file
} directp;
```

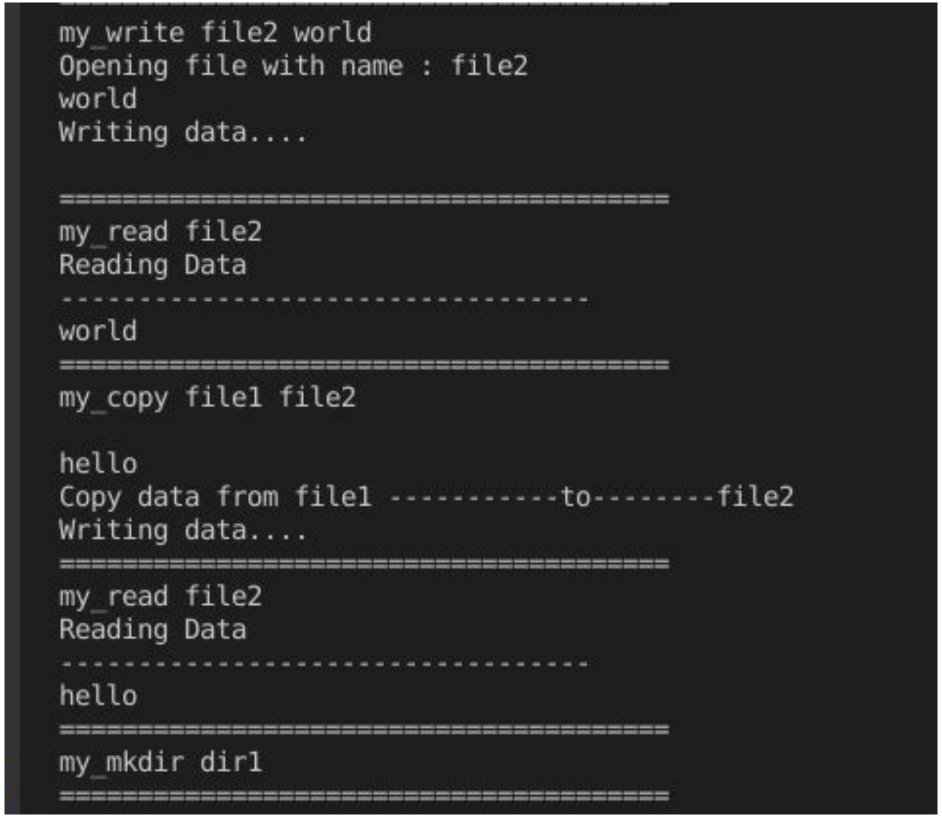


```
typedef struct singleip
{
    directp *sidp[62] = {NULL};
    int size;
    int blockno;
    bool is_dir_or_file;
} singleip;

typedef struct doublyip
{
    singleip *dip[62] = {NULL};
    int size;
    bool is_dir_or_file;
    int blockno;
} doublyip;
```

Outputs



```
----------Enter The Command------------
my_open file1
File created/opened.
The name of the file is file1 with inode 0
=======================================
my_open file2
File created/opened.
The name of the file is file2 with inode 1
=======================================
my_write file1 hello
Opening file with name : file1
hello
Writing data....

=======================================
my_read file1
Reading Data
----------------------------------
hello
=======================================
my_close file1
File file1 closed
=======================================
my_read file1
File is closed, please open it for reading.
=======================================
```



```
my_write file2 world
Opening file with name : file2
world
Writing data....

=======================================
my_read file2
Reading Data
----------------------------------
world
=======================================
my_copy file1 file2

hello
Copy data from file1 -----------to--------file2
Writing data....
=======================================
my_read file2
Reading Data
----------------------------------
hello
=======================================
my_mkdir dir1
=======================================
```