```cpp
// Jacob Bracey
// Thomas Gibbons

#include <iostream>
#include <cstring>
#include <cstdio>
#include <cstdlib>

using namespace std;

// --------- BaseSig class and methods -----------------------------
class BaseSig{
    private:
        // neither derived classes nor other users
        // can access private members
        int readFile(char*);    //NEW method used to fill data

    protected:  // accessible by derived classes, not by other users.
        int length;
        int *raw_data;
        int max;    //NEW variable to hold max of data file
        //int min;  //NEW variable to hold min of data file

    public:
        BaseSig();      // default constructor.
        BaseSig(char*); // parametric constructor
        ~BaseSig();     // destructor
        int getLength() { return length; };
        int getRawValue(int pos);
        static int numObjects;  // static, only one member for the entire hierarchy
        virtual void printInfo();
};

int BaseSig::numObjects = 0;    // initialize static data member

// Base class constructor
BaseSig::BaseSig(){
    length = 0;
    raw_data = NULL;
    numObjects++;
}

// Base class parametric constructor
// Note that the data array is not being initialized (we could read from a file)
BaseSig::BaseSig(char* filename){   //MODIFIED from int to char* for filename
    //length = L;   //MODIFIED
    //raw_data = new int[L];    //MODIFIED
    //if(raw_data == NULL)
    //  cerr << "Error in memory allocation";
    readFile(filename);
    numObjects++;
}
```

```
// Base class destructor
BaseSig::~BaseSig(){
    delete[] raw_data;
    cout << "Goodbye, BaseSig." << endl;
}

int BaseSig::getRawValue(int pos) {
    if(pos < 0)          // invalid index
        return(raw_data[0]);
    else if(pos >= length)  // invalid index
        return(raw_data[length-1]);
    else
        return(raw_data[pos]);
}

int BaseSig::readFile(char* filename){  //new method
    FILE *fp;
    fp=fopen(filename,"r");

    if(fp==NULL)    //check if file is valid
    {
        std::cout << std::endl << filename << "could not be accessed\n";
        return 1;
    }

    fscanf(fp,"%d %d",&length,&max);    //scan info from raw data
    int tempCount=length;   //counter variable
    int x=0;    //counter variable

    raw_data=new int[length];   //allocate memotry for array
    if(raw_data == NULL)
        cerr << "Error in memory allocation";
    else{
        while(tempCount>0)  //fill array
        {
            fscanf(fp,"%d", &raw_data[x]);
            x++;
            tempCount--;
        }
    }
    fclose(fp);

    return 0;
}

void BaseSig::printInfo() {
    cout << "\nBaseSig Info" << endl
         << "Length: " << length << endl;
}
// ----------------------------------------------------------------

// --------- ExtendSig class and methods ---------------------------
class ExtendSig : public BaseSig{ // ExtendSig is derived from class BaseSig
```

```
//BaseSig is a public base class
    private:
        double average;     // add new data members
        double *data;

    public:
        ExtendSig(char*);   //derived classes need a new constructor
        ~ExtendSig();

        // define new member functions
        double getValue(int pos);
        int setValue(int pos, double val);
        double getAverage();


        // redefine member function. Virtual keyword not needed
        void printInfo();   // new standard: explicit "override" keyword can be used
};

// Derived class constructor. Note how the Base constructor is called.
ExtendSig::ExtendSig(char* filename) : BaseSig(filename) {  //MODIFIED from int to char*
for filename
    data = new double[length];
    if(data == NULL)
        cerr << "Error in memory allocation";
    else{
        for(int i = 0; i < length; i++)
            data[i] = (double)raw_data[i];
        average = getAverage();
    }
}

// Derived class destructor
ExtendSig::~ExtendSig() {
    //delete raw_data;
    delete data;
    cout << "Goodbye, ExtendSig." << endl;
}

double ExtendSig::getValue(int pos) {
    if(pos < 0)          // invalid index
        return(data[0]);
    else if(pos >= length)  // invalid index
        return(data[length-1]);
    else
        return(data[pos]);
}

int ExtendSig::setValue(int pos, double val) {
    if((pos < 0) || (pos >= length))
        return(-1); // invalid index
    else {
        data[pos] = val;
```

```
        average = getAverage();
        return(0);  // success
    }

}

double ExtendSig::getAverage() {
    if(length == 0)
        return(0.0);
    else {
        double temp = 0.0;
        for(int i = 0; i < length; i++)
            temp += data[i];
        return(temp/(double)length);
    }
}

// Redefined printInfo function for derived class
void ExtendSig::printInfo() {
    cout << "\nExtendSig Info" << endl
         << "Length: " << length << endl
         << "Average: " << average << endl;
}

// ---------------------------------------------------------------------

class ProcessedSignal : public BaseSig{
    private:
        double average;     // add new data members
        double *data;
        double max;     //NEW
        double min;     //NEW
    public:
        ProcessedSignal(char*); //derived classes need a new constructor
        ~ProcessedSignal();

        // define new member functions
        double getValue(int pos);
        int setValue(int pos, double val);
        double getAverage();
        void max_min();     //NEW
        void scaleFile(double scale);   //NEW
        void normalizeFile(){scaleFile(1.0/max);}   //NEW
        // redefine member function. Virtual keyword not needed
        void printInfo();   // new standard: explicit "override" keyword can be used
};

ProcessedSignal::ProcessedSignal(char* filename) : BaseSig(filename) {  //MODIFIED from int
to char* for filename
    data = new double[length];
    if(data == NULL)
        cerr << "Error in memory allocation";
    else{
```

```
            for(int i = 0; i < length; i++)
                data[i] = (double)raw_data[i];
            average = getAverage();
            max_min();
        }
    }

    ProcessedSignal::~ProcessedSignal() {
        //delete raw_data;
        delete data;
        cout << "Goodbye, ProcessedSignal." << endl;
    }

    double ProcessedSignal::getValue(int pos) {
        if(pos < 0)          // invalid index
            return(data[0]);
        else if(pos >= length)  // invalid index
            return(data[length-1]);
        else
            return(data[pos]);
    }

    int ProcessedSignal::setValue(int pos, double val) {
        if((pos < 0) || (pos >= length))
            return(-1); // invalid index
        else {
            data[pos] = val;
            average = getAverage();
            max_min();
            return(0);  // success
        }
    }

    double ProcessedSignal::getAverage() {
        if(length == 0)
            return(0.0);
        else {
            double temp = 0.0;
            for(int i = 0; i < length; i++)
                temp += data[i];
            return(temp/(double)length);
        }
    }

    void ProcessedSignal::max_min(){
    /*  input:  integer array
                number of integers in array
        updates: max and min values in array*/

        int tempCount=length;
        max=data[0];
        min=data[0];
        while(tempCount>0)
```

```
    {
        max=(max>data[length-tempCount])? max:data[length-tempCount];
        min=(min<data[length-tempCount])? min:data[length-tempCount];
        tempCount--;
    }
}

// Redefined printInfo function for derived class
void ProcessedSignal::printInfo() {
    cout << "\nProcessedSignal Info" << endl
         << "Length: " << length << endl
         << "Average: " << average << endl
         << "Max: " << max << endl
         << "Min: " << min << endl;
}

void ProcessedSignal::scaleFile(double scale){
/*  input:  value of scale
    output: store alteredData*/

    int x=0;
    int count=length;

    //scale each value
    while (count>0)
    {
        data[x]*=scale;
        x++;
        count--;
    }

    //update after changes
    max_min();
    average=getAverage();
}

class ProcessedSignal_v2 : public ExtendSig{
    private:
        double average;     // add new data members
        double *data;
        double max;     //NEW
        double min;     //NEW

    public:
        ProcessedSignal_v2(char*);  //derived classes need a new constructor
        ~ProcessedSignal_v2();

        void scaleFile(double scale);   //NEW
        void normalizeFile(){scaleFile(1.0/max);}   //NEW
        // redefine member function. Virtual keyword not needed
        void printInfo();   // new standard: explicit "override" keyword can be used
        void max_min();     //NEW
```

```
};

ProcessedSignal_v2::ProcessedSignal_v2(char* filename) : ExtendSig(filename) {  //MODIFIED
from int to char* for filename
    data = new double[length];
    if(data == NULL)
        cerr << "Error in memory allocation";
    else{
        for(int i = 0; i < length; i++)
            data[i] = (double)raw_data[i];
        average = getAverage();
        max_min();
    }
}

ProcessedSignal_v2::~ProcessedSignal_v2() {
    //delete raw_data;
    delete data;
    cout << "Goodbye, ProcessedSignal_v2." << endl;
}

// Redefined printInfo function for derived class
void ProcessedSignal_v2::printInfo() {
    cout << "\nProcessedSignal_v2 Info" << endl
         << "Length: " << length << endl
         << "Average: " << average << endl
         << "Max: " << max << endl
         << "Min: " << min << endl;
}

void ProcessedSignal_v2::scaleFile(double scale){
/*  input:  value of scale
    output: store alteredData*/

    int x=0;
    int count=length;

    //scale each value
    while (count>0)
    {
        data[x]*=scale;
        x++;
        count--;
    }

    //update after changes
    max_min();
    average=getAverage();
}

void ProcessedSignal_v2::max_min(){
/*  input:  integer array
```

```
            number of integers in array
    updates: max and min values in array*/

    int tempCount=length;
    max=data[0];
    min=data[0];
    while(tempCount>0)
    {
        max=(max>data[length-tempCount])? max:data[length-tempCount];
        min=(min<data[length-tempCount])? min:data[length-tempCount];
        tempCount--;
    }
}

// Main function. A few examples
int main(){
    //strings to hold the input file names
    char str1[25]="Raw_data_01.txt";
    char str2[25]="Raw_data_02.txt";
    char str3[25]="Raw_data_03.txt";
    char str4[25]="Raw_data_04.txt";

    BaseSig bsig1(str1);
    ExtendSig esig1(str2);

    cout << "# of objects created: " << bsig1.numObjects << endl
         << "# of objects created: " << esig1.numObjects << endl;

    ProcessedSignal psig1(str3);

    cout << "# of objects created: " << psig1.numObjects << endl;

    ProcessedSignal_v2 psigv2(str4);

    cout << "# of objects created: " << psigv2.numObjects << endl;

    //calls all printInfo methods
    bsig1.printInfo();
    esig1.printInfo();
    psig1.printInfo();
    psigv2.printInfo();

    //calls the 2 normalize file methods.
    psig1.normalizeFile();
    psigv2.normalizeFile();

    cout << "-----------------------------------------" << endl;
    cout<<"\nAfter normalize File ran\n";
    psig1.printInfo();
    psigv2.printInfo();

    cout << "-----------------------------------------" << endl;
    //calls get value methods, calls set value method, and then displays the change
```

```
    cout <<endl<<"Values of raw_data and data before update"<<endl<< psig1.getRawValue(7) <<
     endl
         << psig1.getValue(7) << endl;  //shows the original value and normalized value

    cout << endl << psig1.setValue(7, 2.5) << endl; //changes the value in normalized data

    cout <<endl<<"Values of raw_data and data after update"<<endl<< psig1.getRawValue(7) <<
    endl
         << psig1.getValue(7) << endl;
    psig1.printInfo();  //new max since 2.5 is higher than the normalized 1
    cout << "-----------------------------------------" << endl;

    BaseSig *ptrB = &bsig1; // pointer points to object of base class
    BaseSig &refB = bsig1;  // reference to object of base class
    ptrB->printInfo();      // which version is used?
    refB.printInfo();       // which version is used?

    ptrB = &psig1;  // pointer points to the base part of the object of derived class
    BaseSig &refB2 = psig1; // reference bound to the base part of psig1
    ptrB->printInfo();      // which version is used?
    refB2.printInfo();      // which version is used?

    ptrB = &psigv2; // pointer points to the base part of the object of derived class
    BaseSig &refB3 = psigv2; // reference bound to the base part of psigv2
    ptrB->printInfo();      // which version is used?
    refB3.printInfo();      // which version is used?
    cout << "-----------------------------------------" << endl;
    return 0;
}
```