```
//*****************************************************************************************
**
//
//              File:                                   results.h
//
//              Student:                                Sean Herrick
//
//              Assignment:                             Program #11
//
//              Course Name:                    Data Structures II
//
//              Course Number:                  COSC 3100-01
//
//              Due:                                    Deember 8th, 2022
//
//
//              This program is an example of 6 different sorting methods
//
//              Other files required:
//                      1.      sortCompares.cpp
//
//*****************************************************************************************
**

#ifndef RESULTS_H
#define RESULTS_H

//*****************************************************************************************
**

struct Results
{
    int ordCompares,
        unOrdCompares,
        revOrdCompares,
        avgCompares,
        ordCopies,
        unOrdCopies,
        revOrdCopies,
        avgCopies;

    Results ( );

};

//*****************************************************************************************
**

Results :: Results ( )
{
    ordCompares = 0;
    unOrdCompares = 0;
    revOrdCompares = 0;
    ordCopies = 0;
    unOrdCopies = 0;
    revOrdCopies = 0;
}

//*****************************************************************************************
**

#endif
```

```
//*****************************************************************************************
**
//
//              File:                                   sortCompares.cpp
//
//              Student:                                Sean Herrick
//
//              Assignment:                             Program #11
//
//              Course Name:                    Data Structures II
//
//              Course Number:                  COSC 3100-01
//
//              Due:                                    Thursday, December 8th, 2022
//
//
//              This program is an example of 6 different sorting methods
//
//              Other files required:
//                      1.      Results.h
//
//*****************************************************************************************
**


#include <iostream>
#include <iomanip>
#include <fstream>
#include <utility>
#include <string>

using namespace std;

#include "Results.h"

//*****************************************************************************************
**

void getData ( int list [ ], int size, const char filename [ ] );
void insertSort ( int list [ ], int size, int& comp, int& cpy );
void selectSort ( int list [ ], int size, int& comp, int& cpy );
void bubbleSort ( int list [ ], int size, int& comp, int& cpy );
void shellSort ( int list [ ], int size, int& comp, int& cpy );
void heapSort ( int list [ ], int size, int& comp, int& cpy );
void _siftUp ( int list [ ], int size, int& comp, int& cpy );
void _siftDown ( int list [ ], int first, int size, int& comp, int& cpy );
void quickSort ( int list [ ], int size, int& comp, int& cpy );
void _quickSort ( int list [ ], int left, int right, int& comp, int& cpy );
void putMedianLeft ( int list [ ], int left, int right, int& comp, int& cpy );
int partition ( int list [ ], int left, int right, int& comp, int& cpy );
void calcResults ( Results& result );
void displayResults ( Results iResults, Results sResults, Results bResults, Results shResults,
                      Results hResults, Results qResults );

//*****************************************************************************************
**

int main ( )
{
    int ordered [ 1000 ],
        unOrdered [ 1000 ],
        reversed [ 1000 ];

    Results iResults,
```

```
        sResults,
        bResults,
        shResults,
        hResults,
        qResults;

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    insertSort ( ordered, 1000, iResults.ordCompares, iResults.ordCopies );
    insertSort ( unOrdered, 1000, iResults.unOrdCompares, iResults.unOrdCopies );
    insertSort ( reversed, 1000, iResults.revOrdCompares, iResults.revOrdCopies );
    calcResults ( iResults );

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    selectSort ( ordered, 1000, sResults.ordCompares, sResults.ordCopies );
    selectSort ( unOrdered, 1000, sResults.unOrdCompares, sResults.unOrdCopies );
    selectSort ( reversed, 1000, sResults.revOrdCompares, sResults.revOrdCopies );
    calcResults ( sResults );

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    bubbleSort ( ordered, 1000, bResults.ordCompares, bResults.ordCopies );
    bubbleSort ( unOrdered, 1000, bResults.unOrdCompares, bResults.unOrdCopies );
    bubbleSort ( reversed, 1000, bResults.revOrdCompares, bResults.revOrdCopies );
    calcResults ( bResults );

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    shellSort ( ordered, 1000, shResults.ordCompares, shResults.ordCopies );
    shellSort ( unOrdered, 1000, shResults.unOrdCompares, shResults.unOrdCopies );
    shellSort ( reversed, 1000, shResults.revOrdCompares, shResults.revOrdCopies );
    calcResults ( shResults );

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    heapSort ( ordered, 1000, hResults.ordCompares, hResults.ordCopies );
    heapSort ( unOrdered, 1000, hResults.unOrdCompares, hResults.unOrdCopies );
    heapSort ( reversed, 1000, hResults.revOrdCompares, hResults.revOrdCopies );
    calcResults ( hResults );

    getData ( ordered, 1000, "ordered.txt" );
    getData ( unOrdered, 1000, "unordered.txt" );
    getData ( reversed, 1000, "reversed.txt" );
    quickSort ( ordered, 1000, qResults.ordCompares, qResults.ordCopies );
    quickSort ( unOrdered, 1000, qResults.unOrdCompares, qResults.unOrdCopies );
    quickSort ( reversed, 1000, qResults.revOrdCompares, qResults.revOrdCopies );
    calcResults ( qResults );

    displayResults ( iResults, sResults, bResults, shResults, hResults, qResults );

    return 0;
}

//*********************************************************************************************
**

void getData ( int list [ ], int size, const char filename [ ] )
{
    ifstream file;
```

```
    int i = 0;

    file.open ( filename );

    if ( file.fail ( ) )
    {
        cout << "cannot open" << endl << endl;
    }

    else
    {
        while ( ( i < size ) && ( file >> list [ i ] ) )
        {
            i++;
        }

        file.close ( );
    }

}

//*********************************************************************************************
**

void insertSort ( int list [ ], int size, int& comp, int& cpy )
{
    int hold,
        x;

    for ( int i = 1; i < size; i++ )
    {
        hold = list [ i ];
        cpy = ( cpy + 1 );

        for ( x = ( i - 1 ); ( x >= 0 ) && ( ++comp ) && ( list [ x ] > hold ); x-- )
        {
            list [ x + 1 ] = list [ x ];
            cpy = ( cpy + 1 );
        }

        list [ x + 1 ] = hold;
        cpy = ( cpy + 1 );
    }

}

//*********************************************************************************************
**

void selectSort ( int list [ ], int size, int& comp, int& cpy )
{
    int min;

    for ( int i = 0; i < ( size - 1 ); i++ )
    {
        min = i;

        for ( int x = ( i + 1 ); x < size; x++ )
        {
            if ( ( ++comp ) && ( list [ x ] < list [ min ] ) )
            {
                min = x;
            }
        }
```

```
        swap ( list [ min ], list [ i ] );
        cpy = ( cpy + 3 );
    }
}

//*****************************************************************************************
**

void bubbleSort ( int list [ ], int size, int& comp, int& cpy )
{
    bool didSwap = true;

    for ( int i = 0; i < ( size - 1 ) && ( didSwap ); i++)
    {
        didSwap = false;

        for ( int x = ( size - 1 ); ( x > i ); x-- )
        {
            if ( ( ++comp ) && ( list [ x ] < list [ ( x - 1 ) ] ) )
            {
                swap ( list [ x ], list [ ( x - 1 ) ] );
                cpy = ( cpy + 3 );
                didSwap = true;
            }
        }
    }
}

//*****************************************************************************************
**

void shellSort ( int list [ ], int size, int& comp, int& cpy )
{
    int hold,
        x;

    for ( int gap = ( size / 2 ); ( gap >= 10 ); gap = ( gap / 2 ) )
    {
        if ( gap % 2 == 0 )
        {
            gap = ( gap + 1 );
        }

        for ( int i = gap; ( i < size ); i++ )
        {
            hold = list [ i ];
            cpy = ( cpy + 1 );

            for ( x = ( i - gap ); ( x >= 0 ) && ( ++comp ) && ( list [ x ] > hold ); x = ( x - gap )
)
            {
                list [ x + gap ] = list [ x ];
                cpy = ( cpy + 1 );
            }

            list [ x + gap ] = hold;
            cpy = ( cpy + 1 );
        }
    }

    insertSort ( list, size, comp, cpy );
}

//*****************************************************************************************
**
```

```
void heapSort ( int list [ ], int size, int& comp, int& cpy )
{
    for ( int i = 1; i < size; i++ )
    {
        _siftUp ( list, i, comp, cpy );
    }

    for ( int x = ( size - 1 ); x > 0; x-- )
    {
        swap ( list [ x ] ,list [ 0 ] );
        cpy = ( cpy + 3 );
        _siftDown ( list, 0, x, comp, cpy );
    }

}

//*****************************************************************************************
**

void _siftUp(int list[], int child, int& comp, int& cpy)
{
    int parent;

    if (child > 0)
    {
        parent = (child - 1) / 2;

        if ((++comp) && (list[child] > list[parent]))
        {
            swap(list[child], list[parent]);
            cpy += 3;
            _siftUp(list, parent, comp, cpy);
        }
    }
}

//*****************************************************************************************
**

void _siftDown ( int list [ ], int parent, int size, int& comp, int& cpy )
{
    int child;

    child = ( parent * 2 ) + 1;

    if ( child < size )
    {
        if ( ( ( child + 1 ) < size ) && ( ++comp )
                                      && ( list [ ( child ) ] < list [ ( child + 1 ) ] ) )
        {
            child = ( child + 1 );
        }

        if ( ( ++comp ) && ( list [ parent ] < list [ child ] ) )
        {
            swap ( list [ parent ], list [ child ] );
            cpy = ( cpy + 3 );
            _siftDown ( list, child, size, comp, cpy );
        }
    }
}

//*****************************************************************************************
**
```

```
void quickSort (int list [ ], int size, int& comp, int& cpy )
{
    _quickSort ( list, 0, ( size - 1 ), comp, cpy );
    insertSort ( list, size, comp, cpy );
}

//*****************************************************************************************
**

void _quickSort ( int list [ ], int left, int right, int& comp, int& cpy )
{
    int piviot;

    if ( ( right - left ) >= 10 )
    {
        putMedianLeft ( list, left, right, comp, cpy );
        piviot = partition ( list, left, right, comp, cpy );
        _quickSort ( list, left, ( piviot - 1 ), comp, cpy );
        _quickSort ( list, ( piviot + 1 ), right, comp, cpy );
    }
}

//*****************************************************************************************
**

void putMedianLeft ( int list [ ], int left, int right, int& comp, int& cpy )
{
    int center = ( ( right + left ) / 2 );

    if ( ( ++comp ) && ( list [ left ] < list [ center ] ) )
    {
        swap ( list [ left ], list [ center ] );
        cpy = ( cpy + 3 );
    }

    if ( ( ++comp ) && ( list [ right ] < list [ center ] ) )
    {
        swap ( list [ right ], list [ center ] );
        cpy += 3;
    }

    if ( ( ++comp ) && ( list [ left ] > list [ right ] ) )
    {
        swap ( list [ left ], list [ right ] );
        cpy = ( cpy + 3 );
    }
}

//*****************************************************************************************
**

int partition ( int list [ ], int left, int right, int& comp, int& cpy )
{
    int lte = ( left + 1 ),
        gt = right;


    while ( lte <= gt )
    {

        while ( ( ++comp ) && ( list [ lte ] <= list [ left ] ) )
        {
            lte = ( lte + 1 );
        }
```

```
        while ( ( ++comp ) && ( list [ gt ] > list [ left ] ) )
        {
            gt = ( gt - 1 );
        }

        if ( lte < gt )
        {
            swap ( list [ lte ], list [ gt ] );
            cpy = ( cpy + 3 );
            lte = ( lte + 1 );
            gt = ( gt - 1 );
        }
    }

    swap ( list [ left ], list [ gt ] );
    cpy = ( cpy + 3 );

    return gt;
}

//*****************************************************************************************
**

void calcResults ( Results& result )
{
    result.avgCompares = ( ( result.ordCompares + result.unOrdCompares + result.revOrdCompares ) / 3
);
    result.avgCopies = ( ( result.ordCopies + result.unOrdCopies + result.revOrdCopies ) / 3 );
}

//*****************************************************************************************
**

void displayResults ( Results iResults, Results sResults, Results bResults, Results shResults,
                      Results hResults, Results qResults )
{
    cout << setw ( 60 ) << "Compares / Copies" << endl;
    cout << "Sorts" << setw ( 21 ) << "Ordered List" << setw ( 22 ) << "UnOrdered List"
         << setw ( 22 ) << "Reversed List" << setw ( 18 ) << "Average" << endl << endl;

    cout << right << "Insert:" << setw ( 11 ) << iResults.ordCompares << " / " << iResults.ordCopies
         << setw ( 15) << iResults.unOrdCompares << " / " << iResults.unOrdCopies << setw ( 13 )
         << iResults.revOrdCompares << " / " << iResults.revOrdCopies << setw ( 12 )
         << iResults.avgCompares << " / " << iResults.avgCopies << endl;

    cout << right << "Select:" << setw ( 11 ) << sResults.ordCompares << " / " << sResults.ordCopies
         << setw ( 15 ) << sResults.unOrdCompares << " / " << sResults.unOrdCopies << setw ( 15 )
         << sResults.revOrdCompares << " / " << sResults.revOrdCopies << setw ( 14 )
         << sResults.avgCompares << " / " << sResults.avgCopies << endl;

    cout << right << "Bubble:" << setw ( 11 ) << bResults.ordCompares << " / " << bResults.ordCopies
         << setw ( 18 ) << bResults.unOrdCompares << " / " << bResults.unOrdCopies << setw ( 13 )
         << bResults.revOrdCompares << " / " << bResults.revOrdCopies << setw ( 11 )
         << bResults.avgCompares << " / " << bResults.avgCopies << endl;

    cout << right << "Shell:" << setw ( 12 ) << shResults.ordCompares << " / " << shResults.ordCopies
         << setw ( 14 ) << shResults.unOrdCompares << " / " << shResults.unOrdCopies << setw ( 14 )
         << shResults.revOrdCompares << " / " << shResults.revOrdCopies << setw ( 13 )
         << shResults.avgCompares << " / " << shResults.avgCopies << endl;

    cout << right << "Heap:" << setw ( 13 ) << hResults.ordCompares << " / " << hResults.ordCopies
         << setw ( 14 ) << hResults.unOrdCompares << " / " << hResults.unOrdCopies << setw ( 14 )
         << hResults.revOrdCompares << " / " << hResults.revOrdCopies << setw ( 13 )
         << hResults.avgCompares << " / " << hResults.avgCopies << endl;
```

```
    cout << right << "Quick:" << setw ( 12 ) << qResults.ordCompares << " / " << qResults.ordCopies
         << setw ( 15 ) << qResults.unOrdCompares << " / " << qResults.unOrdCopies << setw ( 15 )
         << qResults.revOrdCompares << " / " << qResults.revOrdCopies << setw ( 14 )
         << qResults.avgCompares << " / " << qResults.avgCopies << endl;
}

//*****************************************************************************************
**

/*
                                        Compares / Copies
Sorts         Ordered List        UnOrdered List         Reversed List           Average

Insert:        999 / 1998         257314 / 258317       499500 / 501498      252604 / 253937
Select:     499500 / 2997         499500 / 2997         499500 / 2997        499500 / 2997
Bubble:        999 / 0            499065 / 768957       499500 / 1498500     333188 / 755819
Shell:        6013 / 12026         20949 / 27513         13781 / 20780        13581 / 20106
Heap:        22462 / 47922         17194 / 28575         15965 / 24948        18540 / 33815
Quick:        8387 / 2760          10983 / 9179           8387 / 4260          9252 / 5399

D:\Data Structures II\Fall 2022\sortCompares(2)\x64\Debug\sortCompares(2).exe (process 9048) exited
with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging-
>Automatically close the console when debugging stops.
Press any key to close this window . . .
*/
```