// FILE: heapsort.cpp
#include <iostream>
using namespace std;

// PROTOTYPES
void heapsort(int A[], int n);
// Precondition: A is an array with at least n components.
// Postcondition: The elements of A have been rearranged so

void heapify(int A[], int n);
// Precondition: A is an array with at least n elements.
// Postcondition: The elements of A have been rearranged so that the
// complete binary tree represented by this array is a heap.

void reheapify(int A[], int n);
// Precondition: n > 0, and A is an array with at least n elements.
// These elements
// form a heap **except** that A[0] may be in an incorrect location.
// Postcondition: The A values have been rearranged so that the first n
// elements of A now form a heap.

int parent(int k) {return (k-1)/2;}
// Precondition: k> 0.
// Postcondition: The function assumes that k is the index of
// an array element, where the array represents a complete binary tree.
// The return value is the index of the parent of node k.

int left_child(int k) {return 2*k + 1;}
// Postcondition: The function assumes that k is the index of
// an array element, where the array represents a complete binary tree.
// The return value is the index of the left child of node k.

int right_child(int k) {return 2*k + 2;}
// Postcondition: The function assumes that k is the index of
// an array element, where the array represents a complete binary tree.
// The return value is the index of the right child of node k.

void swap(int& p, int& q);

int main(){
    const int ARRAY_SIZE = 20;  // Number of elements to be sorted
    int A[ARRAY_SIZE];          // Array of integers to be sorted
    int user_input;             // Number typed by the user
    int number_of_elements;     // How much of the array is used
    int i;                      // Array index

    cout << "Please type up to " << ARRAY_SIZE << " positive integers. ";
    cout << "Indicate the end with a zero." << endl;

    // Read the input numbers
    number_of_elements = 0;
    cin >> user_input;

while ((user_input != 0) && (number_of_elements < ARRAY_SIZE))
{
    A[number_of_elements] = user_input;
    ++number_of_elements;
    cin >> user_input;
}

// Sort the numbers and print the result
heapsort(A, number_of_elements);
cout << endl << "In sorted order, the numbers are: ";
for (i = 0; i < number_of_elements; ++i)
    cout << A[i] << "  ";
cout << endl << endl;
return 0;
}  // end of main

void heapsort(int A[ ], int n)
{
    int unsorted;
    heapify(A, n);
    unsorted = n;
    while (unsorted > 1)
    {
        --unsorted;
        swap(A[0], A[unsorted]);
        reheapify(A, unsorted);
    }
}  // end of heapsort

void heapify(int A[ ], int n)
{
    int i;  // Index of next element to be added to heap
    int k;  // Index of new element as it is pushed upward through the heap
    for (i = 1; i < n; ++i)
    {
        k = i;
        while ((k > 0) && (A[k] > A[parent(k)]))
        {
            swap(A[k], A[parent(k)]);
            k = parent(k);
        }
    }
}  // end of heapify
void reheapify(int A[], int n)
{
    int current;          // Index of the element that is moving down
    int big_child_index;  // Index of the larger child of the element
    bool heap_ok = false; // Will change to true when the heap becomes good.

    current = 0;

    // Note: The loop keeps going while the heap is not okay, and while the
    // current element has at least a left child. The test to see whether the
    // current element has a left child is: left_child(current) < n.
    while (!heap_ok && (left_child(current) < n))
    {
        // Compute the index of the larger child:
        if (right_child(current) >= n)
            // There is no right child, so left child must be largest
            big_child_index = left_child(current);
        else if (A[left_child(current)] > A[right_child(current)])
            // The left child is the bigger of the two children
            big_child_index = left_child(current);
        else
            // The right child is the bigger of the two children
            big_child_index = right_child(current);

        // Check whether the larger child is bigger than the current element.
        // If so, swap the current element with its bigger child and
        // continue; otherwise we are done.
        if (A[current] < A[big_child_index])
        {
            swap(A[current], A[big_child_index]);
            current = big_child_index;
        }
        else
            heap_ok = true;
    }
} // end of reheapify

void swap(int& p, int& q)
{
    int temp = p;
    p = q;
    q = temp;
} // end of swap