```
/*
Jung Kim
CS-102 Programming II
Professor Hamaker
Final
*/

#include <iostream>
#include <math.h>

using namespace std;

/*
1a.
int *p
p = new int[100];
for (int i = 0; i < 100; i++)
    p[i] = -1;

b.
<template type>
type A[30],
     B[30];

for (int i = 0; i < 10; i++)
    B[i] = A[9 - i];

c.
int *p, x;
delete p;
p = x;

d.
<template type>
type *p, *q, x;
p = q = x;
delete p;
p = q = NULL;

e.
struct node
{
int n;
node* p;
}

node n, m;
n.p = m;
m.p = m;
*/

/*
2a.i.
(3, 2, 4) -> (2, 3, 4)
a.ii
(2, 3, 4) -> (2, 3, 4)
a.iii
(4.6, 8.2, 5.0) -> (4.6, 5.0, 8.2)
a.iv
("cat", "dog", "Dog") -> ("Dog", "cat", "dog")

b. Overload of fcnA()
template <class T>
void fcnA(T& a, T& b, T&c, bool (*cmp)(T x, T y))
{
    if (*cmp(a, b))
```

```
67          swap(a, b);
68
69      if (*cmp(b, c))
70          swap(b, c);
71
72      if (*cmp(a, b))
73          swap(a, b);
74  }
75
76  template <class T>
77  bool cmp(T x, T y)
78  {
79      return(x > y);
80  }
81  */
82
83  template <class T>
84  void fcnA(T& a, T& b, T& c)
85  {
86      if (a > b)
87          swap(a, b);
88      if (b > c)
89          swap(b, c);
90      if (a > b)
91          swap(a, b);
92  }
93
94  /*
95  3a.
96  Checkpoint A:
97  a = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
98  p = some unknown integer
99  q = an array of some unknown integers
100
101 Checkpoint B:
102 a = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
103 p = 0
104 q = [81, 64, 49, 36, 25, 16, 9, 4, 1, 0]
105
106 Checkpoint C:
107 a = [0, 1, 4, 9, 16, -1, 36, 49, 64, 81]
108 p = -1
109 q = [81, 64, 49, 36, 25, 16, 9, 4, 1, 0]
110
111 b.
112 'p' is an integer pointer thats currently pointing to a + 5 or a[0+5]
113 which is the fifth index in the array 'a'. Thus if p is deleted, then
114 a[5] will also be deleted and the array 'a' will contain some unknown
115 (and undesired) value for a[5]. In fact, my compiler 'ignores' the
116 line 'delete p' and doesn't delete it because it doesn't understand
117 the command to delete an element in an array of integers.
118
119 c.
120 for (int j = 0; j < 10; j++)
121     q[j] = *(p - (j + 1));
122
123 while (p != a)
124     q[j++] = *(--p);
125 */
126
127 void number3()
128 {
129     int *p, *q, a[10];
130     for (int i = 0; i < 10; i++)
131         a[i] = i * i;
132
```

```
133      p = a + 10;
134      q = new int[10];
135
136      cout << "this is just p " << p << endl
137           << "and this is *p " << *p << endl;
138
139      cout << "checkpoint A" << endl;
140      cout << "a = ";
141      for (int i = 0; i < 10; i++)
142          cout << a[i] << " ";
143
144      cout << "\np = " << *p << endl;
145
146      cout << "q = ";
147      for (int i = 0; i < 10; i ++)
148          cout << q[i] << " ";
149      /*
150      for (int j = 0; j < 10; j++)
151          q[j] = *(p - (j + 1));
152      */
153      int j = 0;
154      while (p != a)
155          q[j++] = *(--p);
156
157      cout << "\n\ncheckpoint B" << endl;
158      cout << "a = ";
159      for (int i = 0; i < 10; i++)
160          cout << a[i] << " ";
161
162      cout << "\np = " << *p << endl;
163
164      cout << "q = ";
165      for (int i = 0; i < 10; i ++)
166          cout << q[i] << " ";
167
168      p = a + 5;
169      *p = -1;
170
171      cout << "\n\ncheckpoint c" << endl;
172      cout << "a = ";
173      for (int i = 0; i < 10; i++)
174          cout << a[i] << " ";
175
176      cout << "\np = " << *p << endl;
177
178      cout << "q = ";
179      for (int i = 0; i < 10; i ++)
180          cout << q[i] << " ";
181  }
182
183  /*
184  4a. Quicksort
185  A[] = {4, 7, -1, 0, -3, -2, 8, 1};
186
187  At checkpoint 1:
188  A = {-2, -3, -1, 0, 7, 4, 8, 1}
189  First (first): 0
190  Last (left - 1): 2
191      Checkpoint 1.2 (inside checkpoint 1):
192      A = {-3, -2, -1}
193      First (first): 0
194      Last (left - 1): -1
195      First (right): 1
196      Last (last): 2
197          Checkpoint 1.3 (inside checkpoint 1.2):
198          A = {-3, -2, -1}
```

```
        First (first): 1
        Last (left - 1): 0
        First (right): 2
        Last (last): 2
First (right): 4
Last (last): 7

At checkpoint 2:
A = {-3, -2, -1, 0, 1, 4, 8, 7}
First (first): 4
Last (left - 1): 4
First (right): 6
Last (last): 7
    At checkpoint 2.1 (inside checkpoint 2):
        First (first): 6
        Last (left - 1): 6
        First (right): 8
        Last (last): 7

b.
              4
            /   \
          -1     7
         /  \      \
       -3    0      8
         \    \
         -2    1
*/

template <class type>
void Quicksort(type A[], int first, int last)
{
    if (first >= last)
        return;
    type mid = A[(first + last)/2];
    int left = first,
        right = last;

    while (left < right)
    {
        while ((left < right) && (A[left] < mid))
            left++;

        while ((left < right) && (A[right] > mid))
            right--;

        if (A[left] > A[right])
            swap(A[left], A[right]);
    }

    while (A[right] == mid)
        right++;

    for (int i = 0; i < last + 1; i++)
        cout << A[i] << " ";
    cout << endl;

    cout << "First (first): " << first << endl
         << "Last (left - 1): " << left - 1 << endl;

    Quicksort(A, first, left - 1);

    cout << "First (right): " << right << endl
        << "Last (last): " << last << endl;

    Quicksort(A, right, last);
```

```
}

/*
5a. Binary Trees
Parent node of Tr[n] = Tr[(n - 1)/2] if before-child
                    or = Tr[(n - 2)/2] if after-child

b.
As a tree, it has relations between the values as parent and child;
however, it cannot delete parts of the tree as it is stored as an
array of integers and it is not possible to delete elements in an
array.

c.i.
pTree<int> pt
pt.pTr[3] would be the address of 6 and pt.*pTr[3] would be the value
6. The value of pt.pTr[4] would be the address of NULL and pt.*pTr[4]
would be NULL.

c.ii.
By storing the tree in an array of pointers, though you can't delete
indexes in the array, you can set pointers to the NULL pointer which,
in effect, is the same as 'deleting' values from the tree. For
example, to delete the value '4', set pt.pTr[4] = NULL, meaning the
pointer that points to the address of '4' now points to NULL.

c.iii. [incomplete]
Search
template <class T>
int pTree::search(T x)
{
    for (int i = 0; i < limit; i++)
        if (x == *pTr[i])
            return i; //this is the value sought or 'where it ought
                      //to be'
        else if (i == limit - 1)
        {
            cout << "Value not found." << endl;
            return;
        }
}


Insert
template <class T>
int pTree::insert(T x)
{
    for (int i = 0; i < limit; i++)
        if (x == *pTr[i])
            break; //because the search function returns an index
                   //instead of a bool value, had to include search
                   //in the insert function to make sure user doesn't
                   //insert an existing value.
        else
        {

        }
}

*/

/*
6a.i. Complex class
To cast double 'v' into a complex number:
complex::complex(double v)
    :x(v), y(0.0){}
```

```
331
332  a.ii.
333  Display modulus and square of complex number 'z' to screen:
334  void complex::dispMod()
335  {
336      double complexmod = modulus();
337      cout << "The modulus of z is: " << complexmod << endl;
338      complex zsqr((x * x - y * y), 2 * x * y);
339      cout << "The square of z is: "
340           << zsqr.x << " + "
341           << zsqr.y << "i" << endl;
342  }
343
344  (in main)
345  complex Z(5.0, 3.0);
346  Z.dispMod();
347
348  Console:
349  The modulus of z is: 5.83095
350  The square of z is: 16 + 30i
351
352  b.
353  Complex conjugate:
354  complex complex::conjugate()
355  {
356      complex zconj(x, -1 * y);
357      return zconj;
358  }
359
360  Operator /=
361  complex complex::operator /=(complex z)
362  {
363      x = (x * z.x + y * z.y)/(z.x * z.x + z.y * z.y);
364      y = (y * z.x - x * z.y)/(z.x * z.x + z.y * z.y);
365      return *this;
366  }
367
368  Operator *=
369  complex complex::operator *=(complex z)
370  {
371      x = x * z.x - y * z.y;
372      y = x * z.y + y * z.x;
373      return *this;
374  }
375
376  Operator *
377  complex operator *(complex z, complex w)
378  {
379      double prodRe, prodIm;
380      prodRe = z.x * w.x - z.y * w.y;
381      prodIm = z.x * w.y + z.y * w.x;
382      return complex(prodRe, prodIm);
383  }
384
385  Operator +
386  complex operator +(complex z, complex w)
387  {
388      double sumRe, sumIm;
389      sumRe = z.x + w.x;
390      sumIm = z.y + w.y;
391      return complex(sumRe, sumIm);
392  }
393
394  c.
395  Operator ==
396  bool operator ==(complex z, complex w)
```

```
{
    return (z.re() == w.re() && z.im() == w.im());
}

Operator !=
bool operator !=(complex z, complex w)
{
    return (z.re() != w.re() or z.im() != w.im());
}

d.
Declared constructor:
If a class does not have any constructors declared, a constructor
will
be automatically created that creates an uninitialized object of the
class type. For the complex class, this is fine because the class is
composed of two simple data types double's, x and y where x is the
real and y is the complex, which can be left uninitialized.
(For example, you can declare double x but not have it set to equal a
specific double value; however if the class had more complex data
types as member variables, ie. other classes, those would require
specific constructors).

Overloaded assignment operator:
Unlike other classes with a variety of both simple and complex data
types, it is not easy to assign values to a class. However, since the
Complex class has only two member variables (double x and y) both of
which are the same data type, there is no need to overload the
assignment operator to assign values in a particular manner. It is
perfectly fine to use the default assignment operator as all that is
happening is assigning two double values to double member variables.

Destructor:
Default destructors already exists for simple data types such as
int's, double's, char's, etc. As these simple data types are not
dynamic in nature or point at other values, it is sufficient enough
for the default destructors to deallocate the memory blocks of these
data types. As the complex class only contains two doubles as its
member values, there is no need to create a destructor.

e.
For complex z:
complex::complex()
    :x(0.0), y(0.0){}

For z = complex(2.1, 0.7):
complex::complex(double vr, double vi)
    :x(vr), y(vi){}

*/

class complex
{
public:
    complex();
    complex(double v);
    complex(double vr, double vi);

    double re(){return x;}
    double im(){return y;}
    double modulus(){return sqrt(x*x + y*y);}

    void setRe(double a){x = a;}
    void setIm(double b){y = b;}
    void dispMod();

```

```
462      bool isZero(){return x == 0.0 && y == 0.00;}
463
464      complex conjugate();
465      complex operator +=(complex z);
466      complex operator *=(complex z);
467      complex operator /=(complex z);
468
469      friend complex operator +(complex z, complex w);
470      friend complex operator *(complex z, complex w);
471
472  private:
473      double x, y;
474  };
475
476  complex operator +(complex z, complex w)
477  {
478      double sumRe, sumIm;
479      sumRe = z.x + w.x;
480      sumIm = z.y + w.y;
481      return complex(sumRe, sumIm);
482  }
483
484  complex operator *(complex z, complex w)
485  {
486      double prodRe, prodIm;
487      prodRe = z.x * w.x - z.y * w.y;
488      prodIm = z.x * w.y + z.y * w.x;
489      return complex(prodRe, prodIm);
490  }
491
492  bool operator ==(complex z, complex w)
493  {
494      return (z.re() == w.re() && z.im() == w.im());
495  }
496
497  bool operator !=(complex z, complex w)
498  {
499      return (z.re() != w.re() or z.im() != w.im());
500  }
501
502  complex::complex()
503      :x(0.0), y(0.0){}
504
505  complex::complex(double v)
506      :x(v), y(0.0){}
507
508  complex::complex(double vr, double vi)
509      :x(vr), y(vi){}
510
511  void complex::dispMod()
512  {
513      double complexmod = modulus();
514      cout << "The modulus of z is: " << complexmod << endl;
515      complex zsqr((x * x - y * y), 2 * x * y);
516      cout << "The square of z is: " << zsqr.x << " + "
517                                     << zsqr.y << "i" << endl;
518  }
519
520  complex complex::conjugate()
521  {
522      complex zconj(x, -1 * y);
523      return zconj;
524  }
525
526  complex complex::operator +=(complex z)
527  {
```

```
    x += z.x;
    y += z.y;
    return *this;
}

complex complex::operator *=(complex z)
{
    x = x * z.x - y * z.y;
    y = x * z.y + y * z.x;
    return *this;
}

complex complex::operator /=(complex z)
{
    x = (x * z.x + y * z.y)/(z.x * z.x + z.y * z.y);
    y = (y * z.x - x * z.y)/(z.x * z.x + z.y * z.y);
    return *this;
}

/*
7a. Hash tables
Rat was inserted before Bug otherwise Bug would be in the index
corresponding to its value (index 39 with H(Bug) = 39); however
because it was inserted into index 40 (the next closest int value
to 39) it is assumed that Rat was placed in the index closest to
its corresponding H value (index 39 is the closest to H(Rat) = 38).

b.
H(Fox) = 40 will be placed in index 43 which is the next closest
available index to index 40.

c.
Index| [37] | [38] | [39] | [40] | [41] | [42] | [43] | [44] |
Key  |  Cat |  Dog |  Bug |  Jay |  Ant |  Fox | NULL | NULL |
H(k) |  37  |  38  |  39  |  40  |  41  |  40  |  NA  |  NA  |

*/

int main()
{
    /*
    #3
    number3();
    cout << endl;
    */

    /*
    #4
    int A[] = {4, 7, -1, 0, -3, -2, 8, 1};
    Quicksort(A, 0, 7);
    */

    /*
    #6
    complex Z(5.0, 3.0);
    Z.dispMod();
    */
    return 0;
}
```