

**COM 1101 Fundamentals of Computer Science**  
**Final Exam: March 17, 1998**  
**Fridman-Noy and Rasala Sections**

Name \_\_\_\_\_ ID Number \_\_\_\_\_

| 1: 15 Pts | 2: 15 Pts | 3: 10 Pts | 4: 20 Pts | 5: 15 Pts | 6: 10 Pts | 7:15 Pts | Total |
|-----------|-----------|-----------|-----------|-----------|-----------|----------|-------|
|           |           |           |           |           |           |          |       |

**Problem 1: Triangle Class (15 Points: 5/3/3/4)**

In this problem, you will build some components of a class to represent a triangle in the plane. Since this class will be based on the built-in struct `Point`, we will recall the functions you need to know about the `Point` struct.

*Functions that deal with point coordinates*

```
// Set the coordinates of P to x and y
void SetPoint(Point& P, short x, short y);

// Get the coordinates of P and store in x and y
void GetPoint(const Point& P, short& x, short& y);

// Make and return a Point with coordinates x and y
Point MakePoint(short x, short y);
```

*Functions that move the current graphics position or draw lines*

```
// Set the current graphics position to P
void MoveTo(const Point& P);

// Draw a line from current graphics position to P
void LineTo(const Point& P);

// Draw a line from P to Q
// Q will become the current graphics position
void DrawLine(const Point& P, const Point& Q);
```

Note also that assignment `P = Q` is valid for the `Point` struct.

Now we will state the details of the triangle problem on the next page.

(A) Begin the definition of class `Triangle`. The private data of the class should consist of three fields of type `Point` specifying the coordinates for the three vertices of a triangle. You may choose the names of these fields. The public section of the class definition should begin with two constructors:

(1) The first constructor should pass three parameters `A`, `B`, `C` of type `Point` by const reference.

(2) The second constructor should pass six `short`'s `x1`, `y1`, `x2`, `y2`, `x3`, `y3` to define the coordinates of the three points and each of these `short`'s should have a default value of zero.

```
class Triangle {  
  
    Point a;  
    Point b;  
    Point c;  
  
public:  
  
    Triangle(const Point& A, const Point& B, const Point& C)  
        : a(A), b(B), c(C) { };  
  
    Triangle(  
        short x1 = 0, short y1 = 0,  
        short x2 = 0, short y2 = 0,  
        short x3 = 0, short y3 = 0)  
    {  
        SetPoint(a, x1, y1);  
        SetPoint(b, x2, y2);  
        SetPoint(c, x3, y3);  
    };  
};
```

(B) Define a member function `Draw()` of class `Triangle` that will draw a triangle object in the graphics window by drawing its three edges.

```
void Draw() const {
    MoveTo(a);
    LineTo(b);
    LineTo(c);
    LineTo(a);
};
```

(C) Define a stand-alone function `Distance` that will compute the distance between two `Point`'s. The header should be:

```
double Distance(const Point& P, const Point& Q);
```

In case you have forgotten the mathematical formula for the distance, here is the formula for the distance between point  $(x_1, y_1)$  and point  $(x_2, y_2)$  in coordinate form:

$$\text{distance} = \text{sqrt}((x_2 - x_1)^2 + (y_2 - y_1)^2)$$

```
double Distance(const Point& P, const Point& Q) {
    short x1, y1, x2, y2, dx, dy;

    GetPoint(P, x1, y1);
    GetPoint(Q, x2, y2);

    dx = x2 - x1;
    dy = y2 - y1;

    return sqrt(dx * dx + dy * dy);
}
```

(D) Define a member function `Perimeter()` of class `Triangle` that will compute the sum of the lengths of the edges of a triangle object.

```
void Perimeter() const {
    return Distance(a, b) + Distance(b, c) + Distance(c, a);
};
```

## Problem 2: Templates (15 Points: 4/4/3/4)

In this problem you will build a template `Pair` class to represent a pair of elements of some type `T`. You will define the data members of the class as well as some of its member functions

(A) Begin the definition of class `Pair`. The class should be defined as a template. The private data of the class should consist of two data members, `x` and `y`, of type `T` which is the template parameter.

```
template <class T>
class Pair {

    T x;
    T y;

public:
```

(B) Write a constructor for an object of class `Pair` that takes two arguments of type `T` and initializes the `Pair` fields with the values of these arguments.

```
    Pair() { const T& X, const T& Y) : x(X), y(Y) { };
```

(C) Write a member function `Print` that would print out the values of the two components of the pair in the following form:

```
(x, y)
```

where `x` is the value of the first component and `y` is the value of the second component. The parentheses and comma must be printed as well as the values.

```
// solution will be more general than that required on exam
// solution assumes that << may be used for items of type T

void Print() (ostream& os = cout) const {
    os << "(" << x << ", " << y << ")";
};
```

(D) Write a member function `Flip` that would swap the value of the two data fields.

```
void Flip() {
    T t = x;
    x = y;
    y = t;
};
```

### Problem 3: Dynamic Memory (10 Points: 1/1/2/4/2)

Assume the declarations:

```
double* P;  
double* A;  
long size;
```

(A) Write the code to make `P` point to a dynamically allocated `double` and set the value of that `double` to 3.14159 .

```
P = new double;           // allocate one item  
*P = 3.14159;           // store data value
```

alternately:

```
P = new double(3.14159); // allocate and call constructor
```

(B) Write the code to deallocate the `double` to which `P` points.

```
delete P;
```

(C) Write the code to:

(1) read `size` from the user

(2) make `A` point to a dynamically allocated array of type `double` with `size` elements.

```
size = RequestLong("Size: ");  
A = new double[size]; // allocate array
```

(D) Write the code to read `size` double precision numbers from the user and store these numbers sequentially in the array to which `A` points. Before reading each number, display a prompt of the form:

```
Number i:
```

where the printed index `i` should be in the range  $1 \leq i \leq \text{size}$ . Notice that what is printed for the user may not be the same as the array index used in the program!

```
for (long i = 0; i < size; i++) {  
    cout << "Number " << (i + 1) << ": ";  
  
    A[i] = RequestDouble();  
}
```

(E) Write the code to deallocate the array to which `A` points.

```
delete [] A;
```

#### Problem 4: Operator overloading and dynamic arrays (20 Points: 4/4/2/5/5)

In this problem you will define a class that implements variable-size arrays of double-precision numbers (dynamic arrays).

(A) Start the class by defining the two private fields of class `DynamicArray`:

`size` holds the size of the array  
`data` points to the dynamically allocated array of `double`'s

```
class DynamicArray {  
  
    long    size;  
    double* data;    // define pointer to hold dynamic address  
  
public:
```

(B) Write a constructor for a `DynamicArray` object. The constructor should take an integer argument `n` specifying the size of the array and should set the `size` field and allocate the memory for the array to which `data` will point. If `n` is negative or zero, set `n` to 1 before setting `size` and doing the dynamic allocation.

```
    DynamicArray(long n) {  
        if (n <= 0) n = 1;  
        size = n;  
        data = new double[size];  
    };
```

(C) Write a destructor for `DynamicArray`.

```
    ~DynamicArray() { delete [] data; };
```

(D) Overload a [] operator as a member function of the `Dynami cArray` class. If the index passed to the operator is less than 0, the element in position 0 of the array should be returned. If the index is greater than the largest valid index, the last element of the array should be returned.

```
// will show the non-const version ... const version is similar
// array element must be returned by reference

double& operator[] (long index) {
    if (index < 0)
        index = 0;                // fix index too small
    else if (index >= size)
        index = size - 1;        // fix index too large

    return data[index];          // return array element
};
```

(E) Write a copy constructor for `Dynami cArray`. Recall that a copy constructor takes another object of the same type as a const reference parameter and creates a copy of that object.

```
Dynami cArray(const Dynami cArray& X) {
    size = X.size;                // copy size
    data = new double[size];      // allocate new array data

    for (long i = 0; i < size; i++)
        data[i] = X.data[i];     // copy array elements
};
```

**Problem 5: Sorting and searching (15 Points: 5/5/5)**

(A) Sort the array

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 7 | 6 | 4 | 9 | 0 | 1 |
|---|---|---|---|---|---|---|---|

using **Selection Sort**. Show the results after each step in the table below. Explain, briefly, what happened next to each row.

| index          | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |                                    |
|----------------|---|---|---|---|---|---|---|---|------------------------------------|
| original array | 5 | 2 | 7 | 6 | 4 | 9 | 0 | 1 |                                    |
|                | 0 | 2 | 7 | 6 | 4 | 9 | 5 | 1 | min element = 0                    |
|                | 0 | 1 | 7 | 6 | 4 | 9 | 5 | 2 | min element = 1                    |
|                | 0 | 1 | 2 | 6 | 4 | 9 | 5 | 7 | min element = 2                    |
|                | 0 | 1 | 2 | 4 | 6 | 9 | 5 | 7 | min element = 4                    |
|                | 0 | 1 | 2 | 4 | 5 | 9 | 6 | 7 | min element = 5                    |
|                | 0 | 1 | 2 | 4 | 5 | 6 | 9 | 7 | min element = 6                    |
|                | 0 | 1 | 2 | 4 | 5 | 6 | 7 | 9 | min element = 7                    |
|                |   |   |   |   |   |   |   |   | array done when next to last moves |

(B) Show the work of one pass of the *partition* function in the **Quicksort** algorithm on the array below:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 7 | 6 | 4 | 9 | 0 | 1 |
|---|---|---|---|---|---|---|---|

Assume that the element in position 3 is chosen as a pivot. So pivot value is 6.

| index          | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |
|----------------|---|---|---|---|---|---|---|---|--|
| original array | 5 | 2 | 7 | 6 | 4 | 9 | 0 | 1 |  |
|                | 5 | 2 | 1 | 6 | 4 | 9 | 0 | 7 | I stops at 2. J stops at 7. Swap.<br>I becomes 3. J becomes 6.   |
|                | 5 | 2 | 1 | 0 | 4 | 9 | 6 | 7 | I stops at 3. J stops at 6. Swap.<br>I becomes 4. J becomes 5.   |
|                | 5 | 2 | 1 | 0 | 4 | 9 | 6 | 7 | I stops at 5. J stops at 4. Since $J < I$ , the partition phase ends.<br>Recursive quicksorts:<br>QuickSort(array, 0, 5);<br>QuickSort(array, 5, 8); |

(C) Given the following array

|                     |          |          |          |          |          |          |          |          |          |          |           |
|---------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|-----------|
| <b>index</b>        | <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> | <b>8</b> | <b>9</b> | <b>10</b> |
| <b>array values</b> | 0        | 4        | 12       | 22       | 30       | 34       | 66       | 67       | 70       | 72       | 74        |

list the array elements that are visited during a Binary Search for number 4

array[5] with value 34

array[2] with value 12

array[1] with value 4 ... stop with success

---

list the array elements that are visited during a Binary Search for number 32

array[5] with value 34

array[2] with value 12

array[3] with value 22

array[4] with value 30 ... stop with failure

---

### Problem 6: Pointer Manipulation (10 Points: 3/3/4)

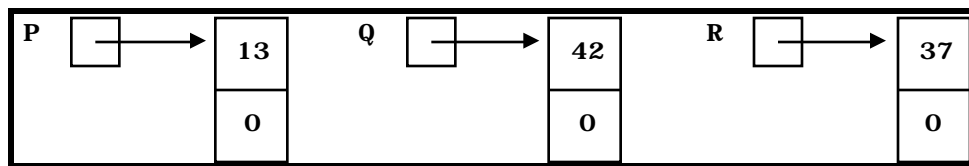
Assume the declarations:

```
struct Node {  
    int    data;  
    Node*  next;  
};
```

```
Node*    P;  
Node*    Q;  
Node*    R;
```

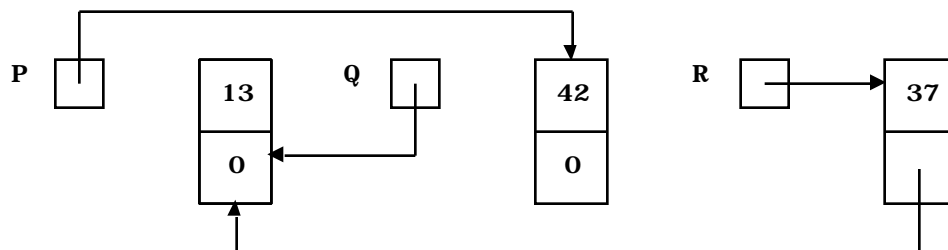
In the exercises below, the NULL pointer will be denoted by 0.

(A) In this part, assume the following initial situation:

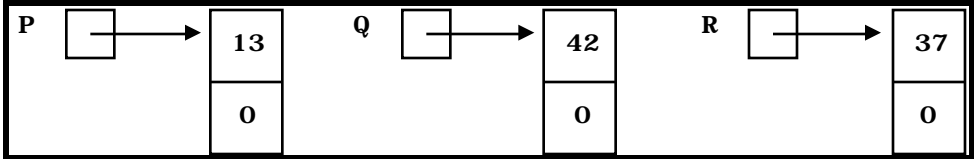


Draw the pointer diagrams after the following code is executed:

```
R->next = P;  
P = Q;  
Q = R->next;
```

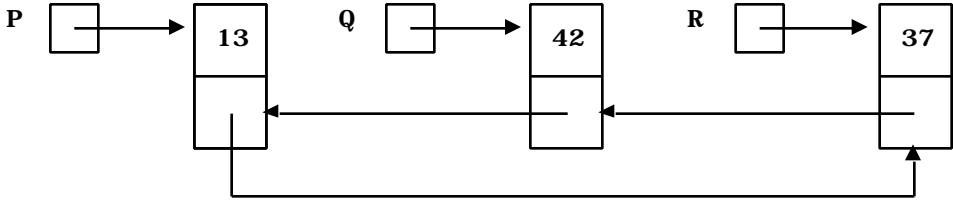


(B) In this part, assume the following initial situation:



Draw the pointer diagrams after the following code is executed:

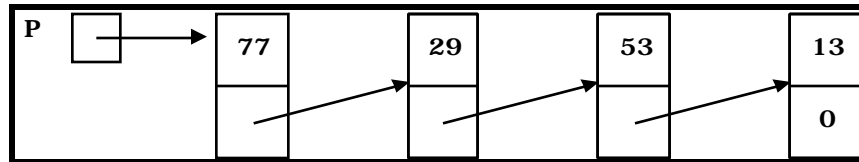
```
P->next = R;  
Q->next = P;  
R->next = Q;
```



(C) In this part, assume the following initial situation:

```
P = 0;
Q = 0;
R = 0;
```

Write the code that will produce the pointer diagram below. Use *dynamic allocation* to obtain the Node's.



You may use pointers Q and R as auxiliary pointers when building the list to which P points but you may not introduce any additional pointer variables.

```

// using only P and starting at the head

P = new Node;
P->data = 77;
P->next = new Node;
P->next->data = 29;
P->next->next = new Node;
P->next->next->data = 53;
P->next->next->next = new Node;
P->next->next->next->data = 13;
P->next->next->next->next = 0;    // null pointer

// using P and Q and starting at the tail

P = new Node;           // create new head at the tail
P->data = 13;           // insert data

Q = P;                 // save old head
P = new Node;           // create new head
P->data = 53;           // insert data
P->next = Q;            // link to old head

Q = P;                 // save old head
P = new Node;           // create new head
P->data = 29;           // insert data
P->next = Q;            // link to old head

Q = P;                 // save old head
P = new Node;           // create new head
P->data = 77;           // insert data
P->next = Q;            // link to old head

// assuming Node has a constructor: Node(int Data, Node* Next)

P = new Node(13, 0); // create first node with null next
P = new Node(53, P); // create new node with link to old head
P = new Node(29, P); // create new node with link to old head
P = new Node(77, P); // create new node with link to old head
  
```

## Problem 7: Singly Linked Lists with Head and Tail (15 Points: 2/3/3/3/4)

Assume the declarations:

```
class Node {
    int    data;
    Node*  next;

public:
    // member functions follow ...
};

class List {
    Node*  Head;
    Node*  Tail;

public:
    // member functions follow ...
}
```

(A) Write a default constructor for class `List` which sets `Head` and `Tail` to null.

```
List() : Head(0), Tail(0) { };
```

(B) Write a constructor for class `Node` which has parameters `theData` and `theNext` and which uses these to set the internal fields `data` and `next`. Arrange that each of the constructor parameters has a default (to zero and null respectively).

```
Node(int theData = 0, Node* theNext = 0)
    : data(theData), next(theNext) { };
```

(C) Write a member function `HeadInsert` of class `List` that will insert at the *head* of the list a new `Node` with data value `item`. Make sure that all pointers are set correctly. The function declaration of `HeadInsert` is as follows:

```
void HeadInsert(int item)

void HeadInsert(int item) {
    Head = new Node(item, Head); // insert node at the head

    if (Tail == 0) Tail = Head; // adjust tail if first node
};
```

(D) Write a member function `TailInsert` of class `List` that will insert at the *tail* of the list a new `Node` with data value `item`. Make sure that all pointers are set correctly.

```
void TailInsert(int item) {
    if (Tail == 0) {
        Head = new Node(item);
        Tail = Head;
    }
    else {
        Tail->next = new Node(item);
        Tail = Tail->next;
    }
};
```

(E) Write a member function `Print` of class `List` that will print each of the integer data items in the list using a width of 12 for each item and printing one item per line.

```
// solution will be more general than that required on exam

void Print(ostream& os = cout) const {
    Node* P = Head;

    // loop as long as P is not null

    while (P) {
        // print data in node
        os << setw(12) << P->data << endl;

        // point to next node
        P = P->next;
    }
};
```