

**COM 1101 Fundamentals of Computer Science**  
**Final Exam: March 16, 1999 Answers**  
**Rasala Sections**

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

1: 14 Pts	2: 20 Pts	3: 14 Pts	4: 16 Pts	5: 18 Pts	6: 18 Pts	Total

**Problem 1: Templates (14 Points: 2/2/3/3/4)**

(A) Define a template `SwapPair` that will swap its two parameters. The parameters `a` and `b` should be of generic type `T` and should be passed by *reference*.

```
template <class T>
void SwapPair(T& a, T& b) {
    T c = a;
    a = b;
    b = c;
}
```

(B) Define a template `SortPair` that will sort its two parameters. The parameters `a` and `b` should be of generic type `T` and should be passed by *reference*. You may assume that the inequality `<` makes sense for type `T`.

```
template <class T>
void SortPair(T& a, T& b) {
    if (b < a)
        SwapPair(a, b);
}
```

(C) Define a template `ArrayFill` that will fill a certain range of cells in the `target` array with a specified `fill` value. The range consists of indices `k` for `lower <= k < upper`.

```
template <class Array1, class T>
void ArrayFill
(Array1& target, long lower, long upper, const T& fill)

{
    for (long k = lower; k < upper; k++)
        target[k] = fill;
}
```

(D) Define a template `ArrayCopy` that will copy a certain range of cells in the `source` array to the `target` array. The range consists of indices `k` for `lower <= k < upper`.

```
template <class Array1, class Array2, class T>
void ArrayCopy
    (Array1& target, const Array2& source, long lower, long upper)

{
    for (long k = lower; k < upper; k++)
        target[k] = source[k];
}
```

(E) Assume `source` is an array of size `M`, `target` is an array of size `N`, and `fill` is a fill value. You will construct a template `ArrayCopyAndFill` that will:

(1) Copy as much of the array `source` into `target` as will fit in `target`.

(2) If `target` is larger than `source` then the remaining cells in `target` will be filled with the value `fill`.

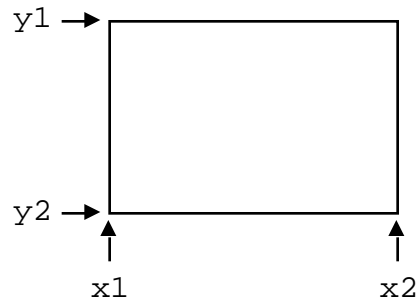
Your code may use `ArrayCopy` and `ArrayFill` to define the template. You must consider the case `M <= N` and the case `M > N` separately.

```
template <class Array1, class Array2, class T>
void ArrayCopyAndFill
    (Array1& target, const Array2& source, long N, long M, const T& fill)

{
    if (M <= N) {
        ArrayCopy(target, source, 0, M);
        ArrayFill(target, M, N, fill);
    }
    else
        ArrayCopy(target, source, 0, N);
}
```

## Problem 2: A Rectangle Class (20 Points: 2/4/2/4/4/2/2)

In this exercise, you will build portions of a `Rectangle` class. This class will deal with a rectangle in screen coordinates (which use `short` integers). You should imagine the rectangle as described geometrically as follows:



Assume that the class definition begins:

```
class Rectangle {
    short x1, y1, x2, y2;

public:
```

(A) We desire to impose a *consistency constraint* that  $x1 \leq x2$  and  $y1 \leq y2$ . Define a member function `Organize()` that will guarantee this constraint by separately sorting the x-values and the y-values. You may use the templates from Problem 1 if they help.

```
    void Organize()
{
    SortPair(x1, x2);
    SortPair(y1, y2);
}
```

(B) Define a `Set` member function that will take 4 short parameters `x1`, `y1`, `x2`, `y2` and set the corresponding member data variables. After this, be sure to *organize* and then return a reference to the object.

```
    Rectangle& Set(short X1, short Y1, short X2, short Y2)
{
    x1 = X1;
    y1 = Y1;
    x2 = X2;
    y2 = Y2;

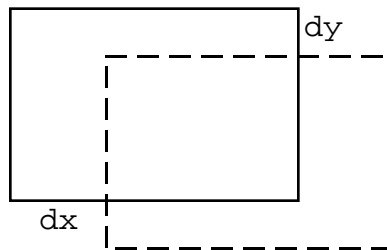
    Organize();

    return *this;
}
```

(C) Define a constructor for class `Rectangle`. The constructor will take 4 short parameters `x1`, `y1`, `x2`, `y2` and should call `Set` to do its work. You should permit `x1`, `y1`, `x2`, `y2` to have default values of zero.

```
Rectangle (short X1 = 0, short Y1 = 0, short X2 = 0, short Y2 = 0) {  
    Set(X1, Y1, X2, Y2);  
}
```

(D) Define a member function `Move(dx, dy)` of class `Rectangle` which should change the internal member data to correspond to a shift by `dx` in the x-direction and `dy` in the y-direction as shown in the diagram below:

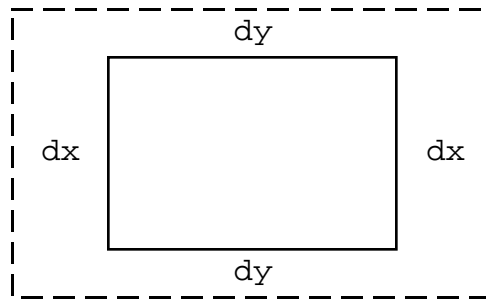


Note that `Move` should not do graphics but should simply modify the coordinates of the `Rectangle` object. The `Move` function should return a reference to the object that is being moved, that is, the object that invokes the `Move`.

The header for `Move` should be:

```
Rectangle& Move(short dx, short dy)  
  
{  
    x1 += dx;  
    y1 += dy;  
    x2 += dx;  
    y2 += dy;  
  
    return *this;  
}
```

(E) Define a member function `Grow(dx, dy)` of class `Rectangle` which should change the internal member data to correspond to expanding the rectangle by `dx` in the x-direction and `dy` in the y-direction as shown in the diagram below:



The `Grow` function should return a reference to the object that invokes the `Grow`.

```
Rectangle& Grow(short dx, short dy) {
    x1 -= dx;
    y1 -= dy;
    x2 += dx;
    y2 += dy;

    Organize();

    return *this;
}
```

(F) Explain by an example why the `Grow` function must call the `Organize` function and then explain why the `Move` function does not need to do this.

In `Grow`, if `dx` or `dy` are large negative values, then the rectangle limits can get out of order and need to be organized. For example, if `x1 = 100`, `x2 = 200`, and `dx = -75`, then `x1` will become 175 and `x2` will become 125 thus requiring a swap. In `Move`, all coordinates are translated by the same amount and so don't require further sorting.

(G) Define a member function `GetLengths(xsize, ysize)` that calculates and returns in `xsize` and `ysize` the lengths of the horizontal and vertical sides of the rectangle. The header should be:

```
void GetLengths(short& xsize, short& ysize) const
{
    xsize = x2 - x1;
    ysize = y2 - y1;
}
```

### Problem 3: Complex Numbers (14 Points: 2/2/2/4/4)

Mathematically, the complex numbers are represented as  $a + b i$  where  $a$  and  $b$  are reals and  $i$  is the square root of  $-1$ . The operations of addition and multiplication are defined as follows:

$$(a + b i) + (c + d i) = (a + c) + (b + d) i$$

$$(a + b i) * (c + d i) = (a * c - b * d) + (a * d + b * c) i$$

In C++, we can represent a complex number simply as a pair  $(x, y)$  of double precision values. The number  $i$  is simply the pair  $(0, 1)$ .

```
class Complex {
    double x;           // real part
    double y;           // imaginary part

public:
```

In this exercise, you will define a constructor and 4 operators for the `Complex` class. Give the headers for the last pair of operators.

(A) Define a constructor for class `Complex` which takes 2 `double` values  $x$  and  $y$  and sets the corresponding member data. Make both parameters have *defaults of zero*.

```
Complex (double X = 0, double Y = 0) : x(X), y(Y) { };
```

(B) Define the friend `operator+` which adds 2 complex numbers  $z$  and  $w$ .

```
friend Complex operator+ (const Complex& Z, const Complex& W)
{
    return Complex(Z.x + W.x, Z.y + W.y);
};
```

(C) Define the member `operator+=` which adds a complex number  $w$  to the object and returns a reference to the object.

```
Complex& operator+= (const Complex& W)
{
    x += W.x;
    y += W.y;

    return *this;
};
```

For convenience, we recapitulate the mathematical definitions:

$$(a + b i) + (c + d i) = (a + c) + (b + d) i$$

$$(a + b i) * (c + d i) = (a * c - b * d) + (a * d + b * c) i$$

(D) Define the friend operator\* which multiplies 2 complex numbers z and w.

```
friend Complex operator* (const Complex& Z, const Complex& W) {
    double X = Z.x * W.x - Z.y * W.y;
    double Y = Z.x * W.y + Z.y * W.x;

    return Complex(X, Y);
};
```

(E) Define the member operator\*= which multiplies the object by a complex number w and returns a reference to the object.

```
Complex& operator*= (const Complex& W) {
    *this = (*this) * W;

    return *this;
};
```

// alternately and more concise:

```
Complex& operator*= (const Complex& W) {
    return (*this = (*this) * W);
}
```

// alternately and more verbose:

```
Complex& operator*= (const Complex& W) {
    double t = x * W.x - y * W.y;    // compute new x but save it

    y = x * W.y + y * W.x;          // compute new y using old x
    x = t;                            // store new x

    return *this;
}
```

#### Problem 4: QuickSort (16 Points: 1/1/2/3/3/4/2)

The source code for one possible implementation of QuickSort is as follows:

```
template <class T>
void QuickSort(T* A, long Lower, long Upper) {
    if (Upper <= Lower + 1)
        return;

    long I = Lower;
    long J = Upper - 1;

    T Pivot = A[ (I + J) / 2 ];

    while (I <= J) {
        while (A[I] < Pivot) I++;           // I-loop
        while (Pivot < A[J]) J--;         // J-loop

        if (I <= J) {                       // swap?
            SwapPair(A[I], A[J]);
            I++;
            J--;
        }
    }

    QuickSort(A, Lower, J + 1);
    QuickSort(A, I, Upper);
}
```

As you see, the array *A* is passed as a pointer. The convention on *Lower* and *Upper* is that the algorithm should sort the elements *A[K]* in the range  $Lower \leq K$  and  $K < Upper$ .

(A) Explain the purpose of the initial lines of code:

```
if (Upper <= Lower + 1)
    return;
```

If the inequality is satisfied then the array range has at most one element and therefore there is no need to sort. This is the base case of the recursion which is essential if the recursion is to halt.

(B) Explain why *J* is initialized to *Upper - 1* rather than to *Upper*.

The last valid index in the range is *Upper - 1* so *J* is initialized to that value.

(C) Explain why the pivot value is copied into a separate variable *Pivot*.

As the algorithm proceeds, the pivot element in the array,  $A[(I + J) / 2]$ , can be swapped into a different position. This will cause us to lose track of the pivot value. To avoid this, the pivot value must be copied to *Pivot*.

(D) Explain the fundamental purpose of the two inner `while` loops. What is accomplished when the loops stop and what can you say about `A[I]` and `A[J]`?

```
while (A[I] < Pivot) I++;           // I-loop
while (Pivot < A[J]) J--;         // J-loop
```

The `I`-loop increases `I` until a large element is found meaning: `A[I] >= Pivot`.  
The `J`-loop decreases `J` until a small element is found meaning: `A[J] <= Pivot`.

If it is still the case that `I <= J` then we should swap `A[I]` and `A[J]`.

(E) After the two inner `while` loops, you see the code:

```
if (I <= J) {                       // swap?
    SwapPair(A[I], A[J]);
    I++;
    J--;
}
```

Why do we need to check if `I <= J`? What could go wrong if we swapped in the case when `I > J`?

If `I > J` then the small element in `A[J]` is already on the left and the large element in `A[I]` is already on the right and it will work against the sort to swap these values.

Why is it absolutely necessary to increment `I` and decrement `J` after the swap?

We must avoid testing `A[I]` and `A[J]` again in this partition phase. If it happens that `A[I]` and `A[J]` both equal the `Pivot` value, then (if we don't update `I` and `J`) we will get stuck with swapping `A[I]` and `A[J]` forever (infinite loop).

(F) Assume that the array A below has 10 elements:

index	0	1	2	3	4	5	6	7	8	9
value	8	1	9	4	3	7	2	0	3	5

In the table below, show the execution of the call `QuickSort(A, 0, 10)` up to the end of the main `while` loop but before the recursive calls. On the right give the new values of `I` and `J` after the `I`-loop, `J`-loop, or swap step (if a swap occurs). In the center of the table show the current state of the array values *after each swap step*. There is no need to show the array values after the `I`-loop and `J`-loop since the values do not change during these loops. For your convenience, cells that do not need to be filled in have been shaded.

The pivot value is 3 which is the value initially in `A[4]`.

index	0	1	2	3	4	5	6	7	8	9	I	J
value	8	1	9	4	3	7	2	0	3	5	0	9
I-loop											0	
J-loop												8
swap?	3	1	9	4	3	7	2	0	8	5	1	7
I-loop											2	
J-loop												7
swap?	3	1	0	4	3	7	2	9	8	5	3	6
I-loop											3	
J-loop												6
swap?	3	1	0	2	3	7	4	9	8	5	4	5
I-loop											4	
J-loop												4
swap?	3	1	0	2	3	7	4	9	8	5	5	3
I-loop												
J-loop												
swap?												

(G) At the conclusion of the execution above, two recursive calls are made. Enter in the blank spaces below the *actual numerical values* that are passed (not symbolic formulas).

```
QuickSort(A,    ,    );
QuickSort(A,    ,    );
```

Answer:

```
QuickSort(A,    0,    4);    // since J + 1 is 4
QuickSort(A,    5,    10);   // since I is 5
```

**Problem 5: Pointer Manipulation for Singly Linked Nodes (18 Points: 3/3/4/4/4)**

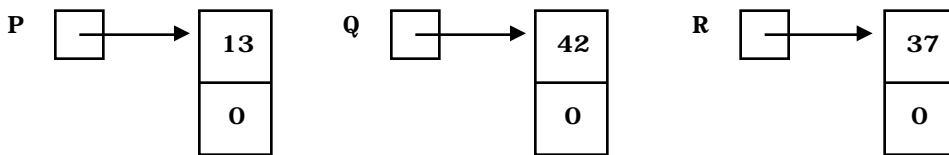
Assume the declarations:

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

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

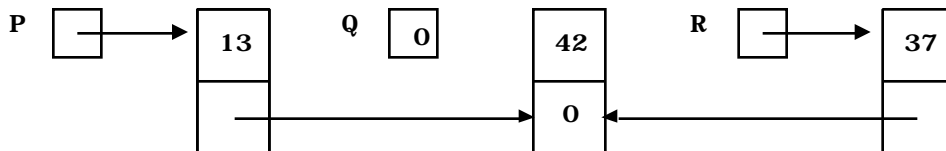
In the exercises below, the null pointer will be denoted by 0.

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

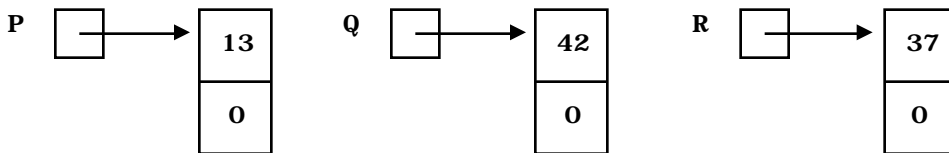


Draw the pointer diagrams after the following code is executed:

```
P->next = Q;
R->next = Q;
Q = 0;
```

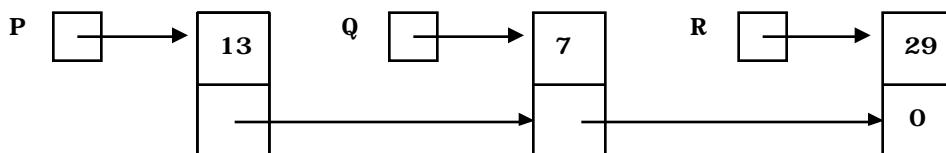


(B) In this part, assume the following initial situation:



Draw the pointer diagrams after the following code is executed:

```
P->next = Q;
P->next->data = 7;
P->next->next = R;
P->next->next->data = 29;
```



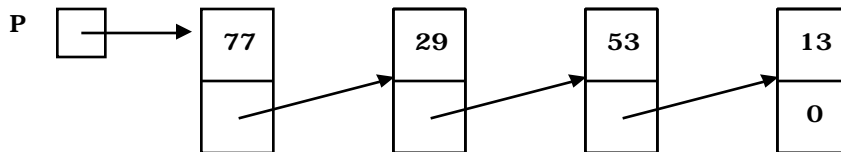
(C) Write a constructor for struct `Node`. This constructor should take 2 parameters. The first called `Data` should be an `int` with default 0 and the second should be a `Node*` pointer called `Next` with a default of null. The corresponding member data should be set.

```
Node (int Data = 0, Node* Next = 0) : data(Data), next(Next) { };
```

(D) 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.

```
P = 0;
P = new Node(13, P);
P = new Node(53, P);
P = new Node(29, P);
P = new Node(77, P);
```

(E) Assume that the linked structure shown in (D) has been constructed using dynamic allocation. Write a loop that will *deallocate* all 4 nodes and then set `P` to null.

```
while (P) {
    Q = P;           // save pointer to old head node
    P = P->next;    // make P point to the next node
    delete Q;      // deallocate old head node
}
```

**Problem 6: Double Play (18 Points: 2/3/2/4/4/3)**

(A) Define an array `A` of 100 `double` values using standard allocation.

```
double A[100];
```

(B) Define a pointer variable `B` that points to type `double`. Using `B`, dynamically allocate an array of 10000 `double` values.

```
double* B;
```

```
B = new double[10000];
```

(C) Deallocate the dynamic memory pointed to by `B`.

```
delete [] B
```

(D) Let `data` represent an array of `double` and let `N` be the number of valid elements in the `data` array. Write a function to compute the average of the valid items in `data`. Use the header:

```
double Average (double* data, long N)
```

To avoid errors, if `N <= 0` then do no calculation and simply return 0.

```
double Average (double* data, long N) {
    if (N <= 0)
        return 0;

    double sum = 0;

    for (long k = 0; k < N; k++)
        sum += data[k];

    return sum/N;
}
```

(E) Let `data` represent an array of `double` and let `N` be the number of valid elements in the `data` array. Let `weight` be an array of `double` containing weight factors that are strictly positive. Write a function to compute the weighed average of the valid items in `data` using the weights in `weight`. Use the header:

```
double WeightedAverage (double* data, double* weight, long N)
```

To avoid errors, if `N <= 0` then do no calculation and simply return 0.

Remark: The weighted average performs a sum of products. Each product multiplies one `data` array item and the corresponding `weight` array item. To find the weighted average, the sum of products (*data times weights*) is divided by the sum of the weights. Hence, two sums must be calculated and then a division must be performed.

```
double WeightedAverage (double* data, double* weight, long N) {
    if (N <= 0)
        return 0;

    double sum_products = 0;
    double sum_weights  = 0;

    for (long k = 0; k < N; k++) {
        sum_products += weight[k] * data[k];
        sum_weights  += weight[k];
    }

    return sum_products / sum_weights;
}
```

(F) Using the notation of part (E), fill the cells of the `weight` array with the following values

1, 1/2, 1/3, 1/4, 1/5, ...

computed as `double` ratios (not `int!`). Then print the weighted average of the `data` using these weights.

```
// in loop, use 1.0 to force double division and offset index k to avoid
// division by 0 when k is 0

for (long k = 0; k < N; k++)
    weight[k] = 1.0 / (k + 1);

cout << WeightedAverage(data, weight, N);
```