

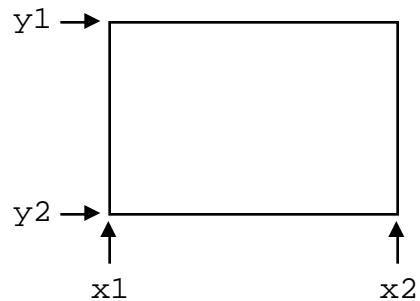
**COM 1101 Fundamentals of Computer Science**  
**Midterm Exam: November 24, 1998**  
**Rasala Section**

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

1: 30 Pts	2: 20 Pts	3: 18 Pts	4: 16 Pts	5: 16 Pts	Total

**Problem 1: A Rectangle Class (30 Points: 2/3/3/4/4/4/2/2/6)**

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:



(A) Begin the definition of class `Rectangle`. Introduce the private data which should consist of four member variables `x1`, `y1`, `x2`, `y2` of type `short`. Start the public section of class `Rectangle` but leave all definitions to later parts of this exercise.

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

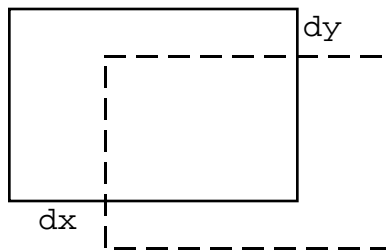
(B) Define a constructor for class `Rectangle`. This constructor should take four parameters `X1`, `Y1`, `X2`, `Y2` of type `short` which should set the corresponding internal member variables. The parameters should not have default values.

```
    Rectangle(short X1, short Y1, short X2, short Y2)  
        : x1(X1), y1(Y1), x2(X2), y2(Y2) { };
```

(C) Define a default constructor for class `Rectangle`. This constructor should take no parameters whatsoever and should set all internal member variables to zero.

```
    Rectangle() : x1(0), y1(0), x2(0), y2(0) { };
```

(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`.

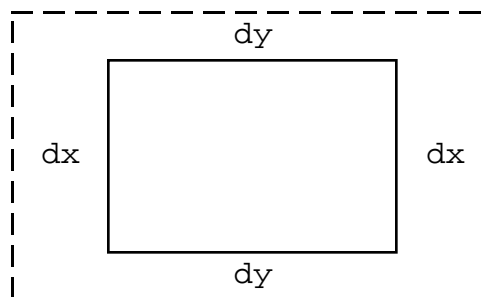
The header for `Move` should be:

```
Rectangle& Move(short dx, short dy)

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;

    return *this;
};
```

(F) Define a member function `Paint()` that will paint a rectangle corresponding to the internal member variables `x1`, `y1`, `x2`, `y2` in the current graphics color. This member function may call the ordinary graphics function `PaintRect` to do its work. The `Paint` function should return a reference to the object that invokes the `Paint`.

```
Rectangle& Paint() const {
    PaintRect(x1, y1, x2, y2);

    return *this;
};
```

(G) For each code sequence below, give the coordinates of the rectangle or rectangles that are painted on the screen. List the coordinates as quadruples: (x1, y1, x2, y2). Also, paint the rectangle or rectangles into the diagram assuming the origin is at the upper left and that the size of each square is 10 pixels.

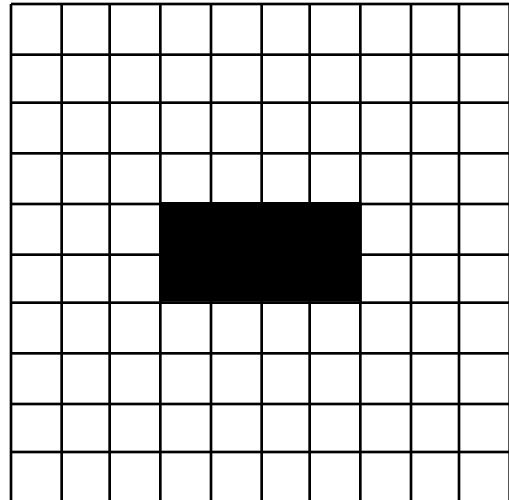
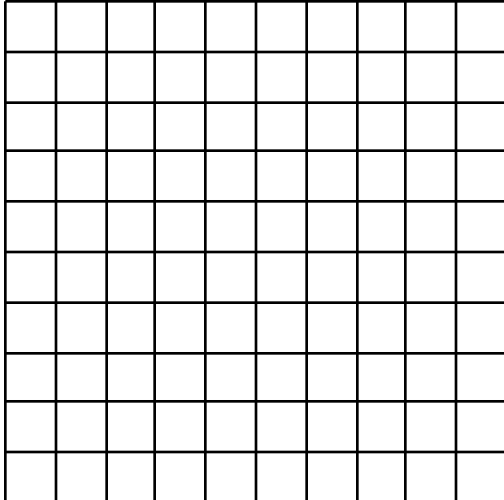
Sequence 1:

```
Rectangle R(20, 10, 60, 30);

R.Move(10, 30);
R.Paint();
```

The rectangle painted is: (     ,     ,     ,     )     The picture is:

*Answer: (30, 40, 70, 60)*



Sequence 2:

**Rectangle S;**

**S. Move(20, 40);**

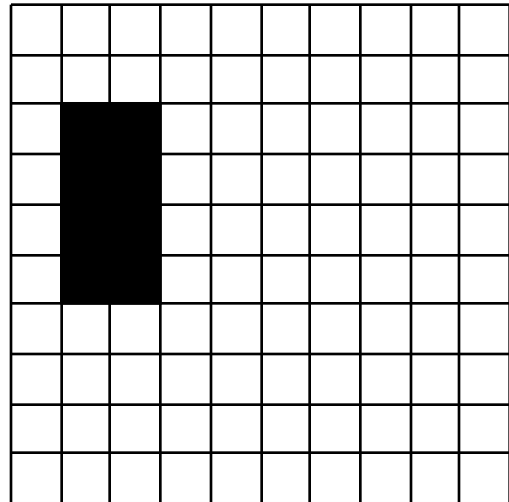
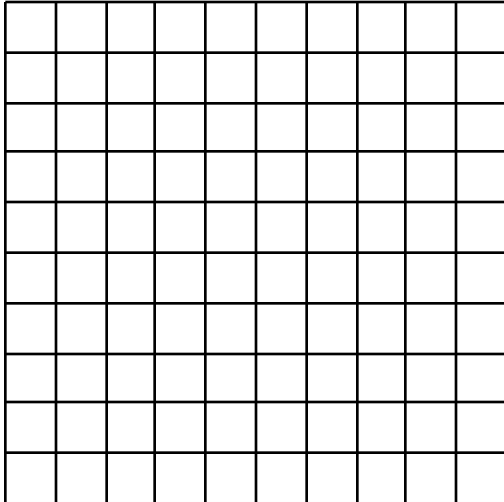
**S. Grow(10, 20);**

**S. Paint();**

The rectangle painted is: ( , , , ).

The picture is:

*Answer:* (10, 20, 30, 60)



Sequence 3:

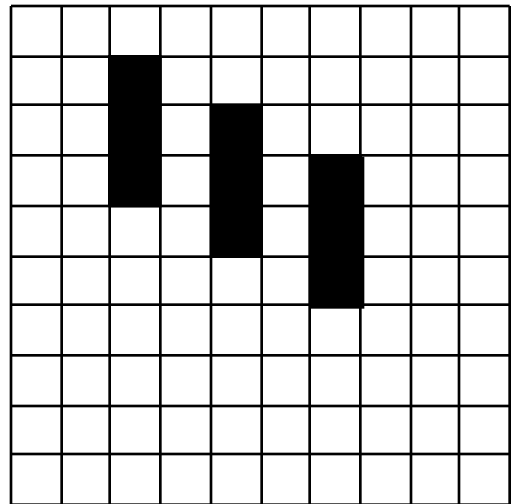
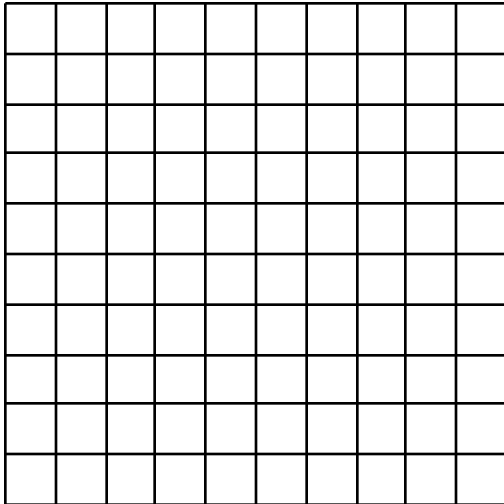
```
Rectangle T(0, 0, 10, 30);
```

```
for (int i = 0; i < 3; i++)  
    T.Move(20, 10).Paint();
```

The rectangles painted are: (     ,     ,     ,     )  
                                  (     ,     ,     ,     )  
                                  (     ,     ,     ,     ).

The picture is:

Answer:     (20, 10, 30, 40)  
              (40, 20, 50, 50)  
              (60, 30, 70, 60)



## Problem 2: Arrays and Pointers (20 Points: 3/3/2/5/5/2)

Assume the declarations:

```
long A[100];
long* B;
long* P;
long size;
```

(A) Write the code to fill **A** with the sequence 0, 3, 6, 9, 12, ... .

```
for (long i = 0; i < 100; i++)
    A[i] = 3*i;
```

(B) Write the code to make **P** point to a dynamically allocated **long** and set the value of that **long** to 144.

```
P = new long;
*P = 144;
```

alternately:

```
P = new long(144);
```

(C) Write the code to deallocate the **long** to which **P** points.

```
delete P;
```

(D) Write the code to:

- (1) read **size** from the user
- (2) make **B** point to a dynamically allocated array of **long** with **size** elements.

```
size = RequestLong("Size: ");
B = new long[size];
```

(E) Using pointer traversal (with **P** as the pointer), write the code to traverse the array to which **B** is pointing and fill that array with the sequence 2, 7, 12, 17, 22, ... .

```
P = B;
long value = 2;

while (P < (B + size)) {
    *P = value;
    P++;
    value += 5;
}
```

(F) Write the code to deallocate the array to which **B** points.

```
delete [] B;
```

### Problem 3: A Wave Function Object (18 Points: 2/2/3/3/5/3)

In mathematics, physics, and engineering, a repetitive or periodic action is often described using the sine or cosine functions. In this exercise, you will define a class **Wave** that can be used to describe such wave functions, specifically, those wave functions of the form:

$$W(t) = a * \cos(2 * \pi * f * (t - c) )$$

Here:

**a = amplitude = maximum height of the wave function**

**f = frequency = number of wave repetitions per unit time**

**c = phase = one particular time value at which  $W(t) == a$**

You will use `operator()` to make a **Wave** object operate as a function.

(A) Begin the definition of class **Wave**. Introduce the private data which should consist of three member variables **a**, **f**, **c** of type **double**.

```
class Wave {
    double a, f, c;

public:
```

(B) Define a default constructor for class **Wave** that sets **a** and **f** to **1** and **c** to **0**.

```
Wave() : a(1), f(1), c(0) { };
```

(C) Define a member function **Set** for class **Wave** that takes three parameters **A**, **F**, **C** of type **double** and sets the corresponding internal variables.

```
void Set(double A, double F, double C) {
    a = A;
    f = F;
    c = C;
};

// note that the specifications did not request that the Set
// function return a reference to the object --- that is why
// there is no code: return *this
```

(D) Define a member function **Get** for class **Wave** that takes three parameters **A**, **F**, **C** of type *reference* to **double** and sets these parameters using the values of the corresponding internal variables.

```
void Get(double& A, double& F, double& C) const {
    A = a;
    F = f;
    C = c;
};
```

(E) You will now define the function operator `operator()` for the class `Wave`. Recall from the previous page that this operator definition should capture the mathematical formula

$$W(t) = a * \cos(2 * \pi * f * (t - c))$$

To simplify your work, we recommend that your operator introduce an internal constant equal to the mathematical constant  $2 * \pi$ , that is:

**6.28318530717958648**

```
double operator() (double t) const {
    const double twopi = 6.28318530717958648;

    return a * cos(twopi * f * (t - c));
};
```

(F) Assume the definition of an object `W`:

```
Wave W;
```

Immediately after this definition, what are the values of the internal variables `a`, `f`, `c` of `W`?

```
a == 1    f == 1    c == 0
```

Next, write the line of code that will set the internal variables `a`, `f`, `c` of `W` to 3, 2, 1 respectively:

```
W.Set(3, 2, 1);
```

Finally, what is now the numerical value of `W(1)`?

```
W(1) == 3 * cos(twopi * 2 * (1 - 1))
      == 3 * cos(0)
      == 3 * 1
      == 3
```

#### Problem 4: Templates and Algorithms (16 Points: 4/4/8)

(A) Write a template function `SwapPair` that will swap the contents of two variables of type `T`.

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

// inline was not specifically mentioned but a small function
// such as SwapPair is usually inline
```

(B) Write a template function `ArrayCopy` that will copy elements from the array `source` into the array `target` over the index range `lower <= i < upper`. The header for this template should be:

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

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

(C) Pick one of the following sorting algorithms: *InsertionSort*, *SelectionSort*, or *QuickSort*. Program this algorithm as a template array algorithm. If you need an auxiliary template then write that as well.

*These algorithms will be handed out on a separate answer sheet.*

### Problem 5: Dates (16 Points: 5/5/6)

(A) Define a function

```
bool IsLeapYear(int year)
```

which returns true if `year` is a leap year and false otherwise. We state the rules that you must implement in this function:

```
    if year is divisible by 400 then year is a leap year
otherwise
    if year is divisible by 100 then year is NOT a leap year
otherwise
    if year is divisible by 4 then year is a leap year
otherwise
    year is NOT a leap year
```

```
bool IsLeapYear(int year) {
    if (year % 400 == 0)
        return true;

    if (year % 100 == 0)
        return false;

    if (year % 4 == 0)
        return true;

    return false;
}
```

(B) Define a function

```
int DaysInMonth(int month, int year)
```

which returns the number of days in a **month** in a given **year**. You may assume that a **month** is an **int** between 1 and 12 and that constants have already been defined to represent each month:

```
jan = 1    feb = 2    mar = 3    ...    dec = 12
```

Remember to deal with leap years in the case of February.

```
int DaysInMonth(int month, int year) {
    switch (month) {
        case feb:
            if (IsLeapYear(year))
                return 29;

            return 28;

        case apr:
        case jun:
        case sep:
        case nov:

            return 30;

        default:

            return 31;
    }
}
```

(C) Assume that a class `Date` is being defined to store information about dates and to handle date manipulations through member functions. Assume that the internal member variables of class `Date` are:

```
int year;
int month;    // month stored as int between 1 and 12
int day;      // day stored as int between 1 and 31
```

Define a member function `Next` of class `Date` such that if `D` is a `Date` then `D.Next()` is the next date in the calendar.

Keep in mind the following examples

```
The next date after April 30, 1998 is May 1, 1998
The next date after February 28, 1999 is March 1, 1999
The next date after February 28, 2000 is February 29, 2000
The next date after December 31, 2000 is January 1, 2001
```

The header for `Next` should be:

```
Date& Next()
```

The `Next` function should return a reference to the date that invokes the `Next`.

```
Date& Next() {
    day++;
    // if ok return
    if (day <= DaysInMonth(month, year))
        return *this;
    // otherwise go to the next month
    day = 1;
    month++;
    // if ok return
    if (month <= 12)
        return *this;
    // otherwise go to the next year
    month = 1;
    year++;
    return *this;
}
```