

COM 1101 Fundamentals of Computer Science
Midterm Exam: February 18, 1999
Rasala Sections

Name _____ ID Number _____ AM or PM? _____

1: 20 Pts	2: 30 Pts	3: 20 Pts	4: 10 Pts	5: 20 Pts	Total

Problem 1: A 3-Dimensional Box Class (20 Points: 5/3/5/3/4)

Introduction: In this problem, we will describe a box by its three dimensions: `length`, `width`, and `height`. We will require that each of these quantities be greater than or equal to zero.

(A) Begin the definition of a 3-dimensional class named `Box`. Introduce the three private variables `length`, `width`, and `height` of type `double`. Then start the public section of the class.

```
class Box {  
    double length;  
    double width;  
    double height;  
  
public:
```

(B) Define a *default constructor* for class `Box` that sets all three member data variables to zero.

```
Box () : length(0), width(0), height(0) { };
```

alternately:

```
Box() {  
    length = 0;  
    width = 0;  
    height = 0;  
};
```

(C) Define a member function `Set` with the following header:

```
Box& Set(double Length, double Width, double Height)
```

This function should:

- (1) Assign the parameters to the corresponding member data variables. *If a parameter is negative then assign the corresponding positive value.*
- (2) Return a reference to the `Box` object.

If you wish, you may use the floating point absolute value function `fabs` to do (1).

```
Box& Set(double Length, double Width, double Height) {  
    length = fabs(Length);  
    width  = fabs(Width);  
    height = fabs(Height);  
  
    return *this;  
};
```

(D) Define a constructor for class `Box` that takes 3 parameters `Length`, `Width`, and `Height` of type `double`. This constructor should call `Set` to do its work.

```
Box(double Length, double Width, double Height) {  
    Set(Length, Width, Height);  
};
```

(E) Define a member function `Volume` which computes the volume of a `Box` object. Use the header:

```
double Volume() const  
  
double Volume() const {  
    return length * width * height;  
};
```

Problem 2: Dates (30 Points: 6/3/6/6/3/6)

Introduction: This problem will define a `Date` class in which the date information is stored using only 2 short integer variables:

`year` the year of the date
`days` the count of days from the first day of the year

For example, the first day of the 1999 would be stored as `year = 1999` and `days = 1`. The last day of 1999 would be stored as `year = 1999` and `days = 365`.

Obtaining information about the month and day of the month will be *calculated* from the `year` and `days` information rather than stored.

Before we can work on this class, we will need some global functions.

(A) Define a function

```
bool IsLeapYear(short 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(short 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 `TotalDays` which return the total number of days in a given year. Of course, this function should return either 365 or 366 as appropriate. The header is:

```
short TotalDays(short year)

short TotalDays(short year) {
    if (IsLeapYear(year))
        return 366;
    return 365;
};
```

(C) Define a function

```
short DaysInMonth(short year, short month)
```

which returns the number of days in a month in a given year. You may assume that a month is a short 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.

```
short DaysInMonth(short year, short month) {
    switch (month) {
        case feb:
            if (IsLeapYear(year))
                return 29;
            return 28;
        case apr:
        case jun:
        case sep:
        case nov:
            return 30;
        default:
            return 31;
    }
}
```

The `Date` class will now be introduced as follows:

```
class Date {  
    short year;  
    short days;    // must satisfy 1 <= days <= TotalDays(year)  
  
public:
```

(D) Define a `Set` member function of class `Date` with the header

```
Date& Set(short Year, short Days)
```

(1) Set the member data variables `year` and `days` after suitably modifying the input parameters `Year` and `Days` to achieve the constraint:

```
1 <= Days <= TotalDays(Year)
```

For example, if `Year` is 1999 and `Days` is 1500, you should perform the following adjustment steps in an appropriate `while` loop:

```
Year = 1999    Days = 1500    // initial value  
Year = 2000    Days = 1500 - 365 = 1135  
Year = 2001    Days = 1135 - 366 = 769    // 2000 is leap!  
Year = 2002    Days = 769 - 365 = 404  
Year = 2003    Days = 404 - 365 = 39
```

A similar `while` loop should adjust in the case when `Days` is initially less than or equal to zero. When the adjustments are done, assign `Year` to `year` and `Days` to `days`.

(2) Return a reference to the `Date` object.

```
Date& Set(short Year, short Days) {  
    while (Days > TotalDays(Year)) {  
        Days -= TotalDays(Year);  
        Year++;  
    }  
  
    while (Days < 1) {  
        Year--;  
        Days += TotalDays(Year);  
    }  
  
    year = Year;  
    days = Days;  
  
    return *this;  
};
```

(E) Write a `Get` member function that will retrieve the internal `year` and `days` information. Use the header:

```
void Get(short& Year, short& Days) const

void Get(short& Year, short& Days) const {
    Year = year;
    Days = days;
};
```

(F) Write a member function `GetYMD` that will *calculate* the `Month` and `MonthDay` and will return that information together with the `Year`. Use the header:

```
void GetYMD(short& Year, short& Month, short& MonthDay) const
```

For example, suppose `year = 2000` and `days = 175`. This function should *calculate* as follows:

```
Year = year = 2000

Month = 1    MonthDay = days = 175
Month = 2    MonthDay = 175 - 31 = 144    // jan has 31 days
Month = 3    MonthDay = 144 - 29 = 115    // feb has 29 days
Month = 4    MonthDay = 115 - 31 = 84     // mar has 31 days
Month = 5    MonthDay = 84 - 30 = 54     // apr has 30 days
Month = 6    MonthDay = 54 - 31 = 23     // may has 31 days
```

So the function returns `Year = 2000`, `Month = 6`, `MonthDay = 23` in this case. Use a `while` loop and take advantage of the `DaysInMonth` function defined earlier.

```
void GetYMD(short& Year, short& Month, short& MonthDay) const {

    Year      = year;
    Month     = 1;
    MonthDay  = days;

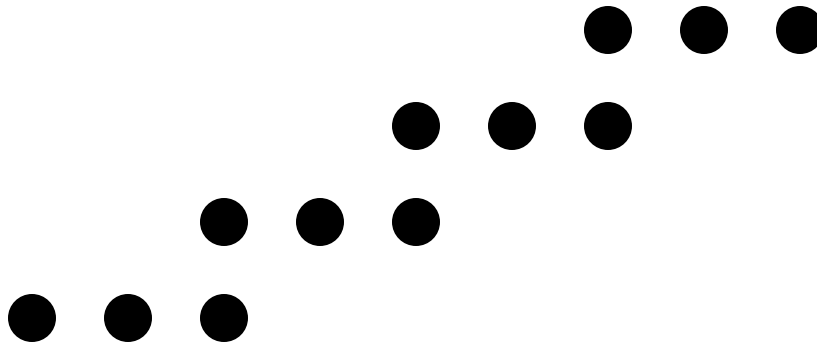
    while (MonthDay > DaysInMonth(Year, Month)) {
        MonthDay -= DaysInMonth(Year, Month);
        Month++;
    }
};
```

Problem 3: Patterns of Circles (20 Points: 10/10)

Introduction: In this problem, you will draw patterns of circles. Each circle will have a radius of 8 pixels. You will use `PaintCircle(x, y, radius)` for appropriate choices of `x`, `y` that will be computed in loops. Keep in mind that `y` increases downward in graphics.

(A) The Staircase

Draw the following staircase pattern in which the initial circle at the lower left has center at (72, 200) and in which the spacing between circles is 32 pixels, either in the horizontal or the vertical direction. Use loops not 12 individual calls!



```
short x = 72;          // initial x position
short y = 200;        // initial y position

for (short row = 0; row < 4; row++) {
    for (short col = 0; col < 3; col++) {
        PaintCircle(x, y, 8);    // paint
        x += 32;                 // move right
    }

    x -= 32;                   // move x back before moving upward
    y -= 32;                   // move upward
}
```


Problem 4: Exponential Growth or Decay (10 Points: 8/2)

In mathematics, exponential growth (such as human population) and exponential decay (such as radioactivity) are described by functions of the form:

$$E(x) = a \cdot e^{bx} = a * \exp(b * x)$$

Growth corresponds to $b > 0$ and decay to $b < 0$. Assume that we have introduced a class to define this family of functions:

```
class Exponential {  
  
    double a;  
    double b;  
  
public:  
  
    Exponential (double A = 1, double B = 1);  
    Exponential& Set(double A, double B);  
    void Get(double& A, double& B) const;  
    double operator() (double x) const;  
};
```

(A) Complete the definition of the “function call operator” `operator()`.

```
double operator() (double x) const {  
    return a * exp(b * x);  
};
```

(B) Assume the sequence of definitions.

```
double x;  
Exponential E;  
  
E.Set(3, 5);  
  
x = E(2);
```

Write down a mathematical expression for the value in x . Don't try to numerically evaluate it ... just give the formula.

$$3 \cdot e^{5 \cdot 2} = 3 \cdot e^{10}$$

Problem 5: A List of Numbers (20 Points: 2/2/6/4/6 = 1.4.1)

Let us introduce a `struct` to hold a list of up to 100 numbers of type `double`.

```
const int maxList = 100;

struct NumberList {
    double list[maxList];
    int listSize;
```

(A) Define a member function `Reset()` that will set `listSize` to zero.

```
void Reset() { listSize = 0; };
```

(B) Define a default constructor that will use `Reset` to set `listSize` to zero.

```
NumberList() { Reset(); };
```

(C) Define a member function `Add` which takes a parameter `x` of type `double` and places `x` into the next empty cell in the `list` and subsequently increments `listSize`. This function should do appropriate error checking.

```
void Add(double x) {
    if (listSize < maxList) {
        list[listSize] = x;
        listSize++;
    }
};
```

(D) Define a member function `Sum` which takes no parameters and returns the sum of the numbers currently in the `list`.

```
double Sum() const {
    double sum = 0;
    for (int i = 0; i < listSize; i++)
        sum += list[i];
    return sum;
}
```

(E) Assume that in another part of our program we are going to use a `NumberList` structure to read, collect, and sum a list of numbers input by the user. Fill in the code for such actions by elaborating on the comments.

```
// Introduce a variable of type NumberList

NumberList L;

// Using ReadingDouble, write a loop that will request and read
// a list of double's from the user until the user hits return.
// These double's should be collected into the NumberList.

double x;

while (ReadingDouble("Enter data: ", x))
    L.Add(x);

cout << endl;

// Print out the sum of the numbers in the list as: Sum = ...

cout << "Sum = " << L.Sum() << endl;
```