

COM 1101 Fundamentals of Computer Science
Final Exam: December 11, 1998
Rasala Section

Name _____ ID Number _____

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

Note that the exam points total to 105 points and therefore build in a bit of extra credit.

Problem 1: Names (20 Points: 2/4/2/2/4/4/2)

In this problem, you will build portions of a class called `Name` that will hold information about a person's name. This class should have five member data components each of type `string` to hold the person's last name, first name, middle name, nickname, and title (such as Mr, Mrs, Miss, Ms, Dr, Prof, etc).

(A) Begin the definition of class `Name` and introduce the five member data components which should be private.

```
class Name {  
    string last, first, middle, nick, title;  
  
public:
```

(B) Define a constructor for class which accepts up to five pieces of `string` data passed by *const reference*. These parameters should set the five member data components: last name, first name, middle name, nickname, and title. The last name should be a required parameter and the other four parameters should have a default of the empty string("").

```
    Name (const string& Last,  
          const string& First = "",  
          const string& Middle = "",  
          const string& Nick = "",  
          const string& Title = "")  
  
    : last(Last), first(First), middle(Middle),  
      nick(Nick), title(Title) { };
```

(C) Show how to construct a `Name` object called `president` that stores the information for William Jefferson Clinton with the nickname of Bill and the title of President.

```
Name president("Clinton", "William", "Jefferson", "Bill", "President");
```

(D) Define a member function `GetLastName` of the class which returns in a *reference parameter* the value of the last name member data and which returns `void` as a function.

```
void GetLastName(string& last) const {
    Last = last;
};
```

(E) Define a member function `SetLastName` of the class which accepts a new last name as a *const reference parameter*, sets the last name member data with this value, and then returns a *reference to the object being acted upon*.

```
Name& SetLastName(const string& Last) {
    last = Last;

    return *this;
};
```

(F) Define a member function

```
void PrintInformal(ostream& out = cout) const
```

that will print the name information to the output stream `out` according to the following rules:

If the nickname is non-empty then print

the nickname, a blank, and the last name

otherwise, if the first name is non-empty then print

the first name, a blank, and the last name

otherwise print

only the last name

You should **not** add a newline (`endl`).

```
void PrintInformal(ostream& out = cout) const {
    if (nick != "")
        out << nick << " " << last;
    else if (first != "")
        out << first << " " << last;
    else
        out << last;
};
```

(G) Give the output printed by the following line of code and tell whether it is printed to a file or to the screen:

```
president.PrintInformal ();
```

Since the call has no parameters, the data is printed to the default output stream `cout` which is the screen. The output itself is:

```
Bi ll Cl int on
```

Problem 2: QuickSort (20 Points: 2/2/2/4/4/4/2)

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	7	3	5	8	6	0	4	9	1	2

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.

index	0	1	2	3	4	5	6	7	8	9	I	J
value	7	3	5	8	6	0	4	9	1	2	0	9
I-loop											0	
J-loop												9
swap?	2	3	5	8	6	0	4	9	1	7	1	8
I-loop											3	
J-loop												8
swap?	2	3	5	1	6	0	4	9	8	7	4	7
I-loop											4	
J-loop												6
swap?	2	3	5	1	4	0	6	9	8	7	5	5
I-loop											6	
J-loop												5
swap?												
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,    6);    // Lower == 0 and J == 5 so J+1 == 6
QuickSort(A,    6,    10);    // I == 6 and Upper == 10
```

Problem 3: Parameter Passage (20 Points: 4/4/4/4/4)

Assume the declarations:

<pre>struct Junk { int x; int y; }; void Foo2(Junk& J) { J.x += 2; J.y += 2; }</pre>	<pre>void Foo1(Junk J) { J.x += 1; J.y += 1; } void Foo3(const Junk& J) { J.x += 3; J.y += 3; }</pre>
<pre>void Foo4(Junk* P) { P->x += 4; P->y += 4; }</pre>	<pre>void Foo5(const Junk* P) { P->x += 5; P->y += 5; }</pre>

(A) Explain the concept of *passage by value*.

In passage by value, a copy is made of the data passed and the function parameter name refers to this copy rather than to the original data. This has two results. Any changes to the function parameter will not change the original data. In addition, it is permissible to pass an expression by value since the expression will be evaluated and stored in the new location associated with the parameter name.

Which of the `Foo` functions use passage by value?: `Foo1`

(B) Explain the concept of *passage by reference*.

In passage by reference, the function parameter name is made into an alias (alternate name) for the original data. Thus no copy is done. Instead, all uses of the function parameter name refer to the original data location. In particular, all changes to the function parameter automatically change the original data since there is only one location in memory that is being referenced.

Passage by reference uses the mechanism of a hidden pointer to accomplish the alias. By using a hidden pointer, the compiler assumes the responsibility for the pointer operations `&` and `*` as needed. Programmer code becomes less cluttered.

Passage by reference is fast for large data structures. To achieve the speed of passage by reference but prevent change to the original, passage by const reference is used.

If `T` is a typename, then passage by reference is `T&` and passage by const reference is `const T&`.

Which of the `Foo` functions use passage by reference?: `Foo2` and `Foo3`

(C) Explain the concept of *passage by pointer*.

In passage by pointer, a pointer to some data type is passed rather than passing the data type itself directly (either by value or by reference).

To pass an ordinary variable *v* as an argument which expects a pointer, you must pass the address of *v*, that is, *&v*.

If *P* is the name of a pointer argument, then to access the associated data you must either use **P* or *P->fieldname*.

In other words, passage by pointer and usage of the pointer require manual coding of the *&*, ***, and *->* operators.

There are four common forms of passage by pointer (where *T* is a typename):

(1) *T* P*

In this case the pointer value is copied and the data may be accessed and modified using either **P* or *P->fieldname*.

(2) *const T* P*

In this case the pointer value is copied and the data may be accessed but *not* modified using either **P* or *P->fieldname*.

(3) *T* & P*

*In this case the pointer is passed by reference so that changes to the pointer itself will be reflected in the original pointer. The data may be accessed and modified using either *P or P->fieldname.*

(4) *const T* & P*

*In this case the pointer is passed by reference so that changes to the pointer itself will be reflected in the original pointer. The data may be accessed but not modified using either *P or P->fieldname.*

Which of the *Foo* functions use passage by pointer?: *Foo4* and *Foo5*

<pre>struct Junk { int x; int y; };</pre>	<pre>void Foo1(Junk J) { J.x += 1; J.y += 1; }</pre>
<pre>void Foo2(Junk& J) { J.x += 2; J.y += 2; }</pre>	<pre>void Foo3(const Junk& J) { J.x += 3; J.y += 3; }</pre>
<pre>void Foo4(Junk* P) { P->x += 4; P->y += 4; }</pre>	<pre>void Foo5(const Junk* P) { P->x += 5; P->y += 5; }</pre>

(D) Some of the above functions produce an error at compile time. Which ones?

Foo3 and Foo5

Explain why these function definitions produce an error.

Foo3 changes the data fields in the `const` parameter `J`.

Foo5 changes the data fields of the object pointed to by the `const` pointer `P`.

(E) For the code sequence below, show the printed output for each `cout` statement and explain the output.

Statements	Output	Output Explanation
<pre>Junk S; Junk* Q = &S;</pre>		Data just assigned so output is same as data
<pre>S.x = 1; S.y = 3; cout << S.x << " " << S.y << "\n";</pre>	1 3	
<pre>Foo1(S); cout << S.x << " " << S.y << "\n";</pre>	1 3	Foo1 operates on copy of data so original is unchanged
<pre>Foo2(S); cout << S.x << " " << S.y << "\n";</pre>	3 5	Foo2 operates on same location as original so data is changed
<pre>Foo4(Q); cout << S.x << " " << S.y << "\n";</pre>	7 9	Foo4 operates on same location as original via <code>-></code> operator so data is changed

Problem 4: The Array Class (20 Points: 5/4/5/6)

In lectures, we introduced a dynamic array class which begins:

```
template <class T> class Array {
    T*    arraydata;    // pointer to the allocated array
    long arraysize;    // size of the allocated array

    // member functions follow ...
}
```

(A) The constructor for the Array class has the header:

```
Array(long Size = 1, const T& Fill = T())
```

Explain in English what tasks the constructor must do.

The constructor must error check `Size` to make sure it is greater than or equal to 1. Then it must store `Size` in the member data `arraysize` and dynamically allocate an array of that size using the member pointer variable `arraydata`. Finally, it must fill the array with the value `Fill`.

Now write the body of the constructor for the Array class.

```
Array(long Size = 1, const T& Fill = T()) {
    if (Size < 1) Size = 1;

    arraysize = Size;
    arraydata = new T[Size];

    for (long i = 0; i < Size; i++)
        A[i] = Fill;
};
```

(B) Explain in English what tasks the destructor of the Array class must do.

The destructor must give back the memory allocated to the array pointed at by `arraydata`.

Now write the destructor of the Array class (header and body).

```
~Array() { delete [] arraydata; };
```

(C) The copy constructor for the Array class has the header:

```
Array(const Array& source)
```

Explain in English what tasks the copy constructor must do.

*The copy constructor must built a new array object that is an exact clone of the array object called **source** but which does not share data with **source**.*

Now write the body of the copy constructor for the Array class.

```
Array(const Array& source) {  
    arraysize = source. arraysize;    // clone size  
    arraydata = new T[arraysize];    // obtain a new array  
  
    // clone source values  
    for (long i = 0; i < arraysize; i++)  
        arraydata[i] = source.arraydata[i];  
};
```

(D) The assignment operator (`operator=`) for the Array class has the header:

```
Array& operator= (const Array& source)
```

Explain in English what tasks the assignment operator (`operator=`) must do.

*The `operator=` must replace the existing array object with an array object that is an exact clone of the array object called **source** but which does not share data with **source**. If necessary (because the array sizes don't match), deallocate the current array data and allocate new array data. Return a reference to the array object to meet C++ requirements for an assignment operator.*

Now write the body of the assignment operator (`operator=`) for the Array class.

```
Array& operator= (const Array& source) {  
    // exit if source and this object are identical  
    if (this == &source)  
        return *this;    // return this object  
  
    // deallocate and reallocate if necessary  
    if (arraysize != source.arraysize) {  
        delete [] arraydata;  
        arraysize = source.arraysize;  
        arraydata = new T[arraysize];  
    }  
  
    // clone source values  
    for (long i = 0; i < arraysize; i++)  
        arraydata[i] = source.arraydata[i];  
  
    return *this;    // return this object  
};
```

Problem 5: Pointer Manipulation for Singly Linked Nodes (10 Points: 3/3/4)

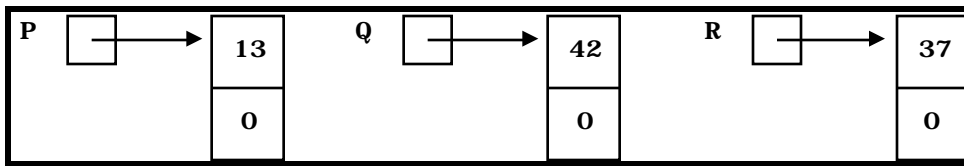
Assume the declarations:

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

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

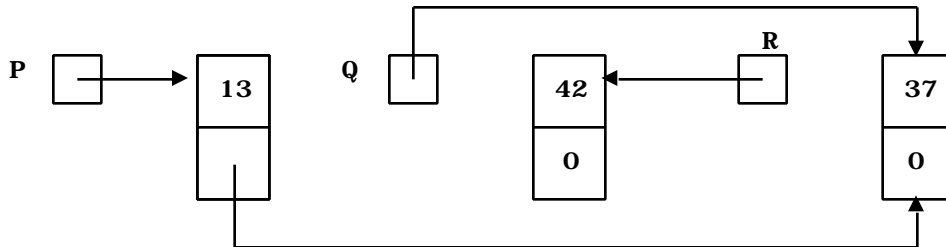
In the exercises below, the NULL pointer will be denoted by 0.

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

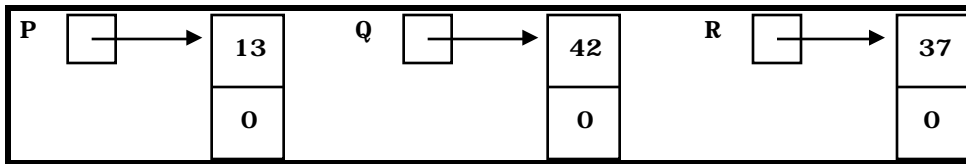


Draw the pointer diagrams after the following code is executed:

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

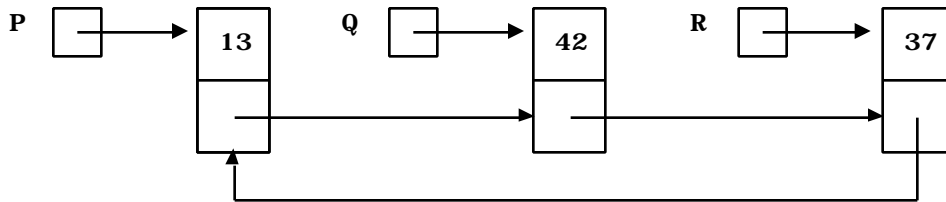


(B) In this part, assume the following initial situation:



Draw the pointer diagrams after the following code is executed:

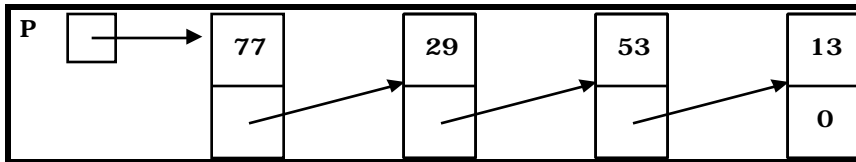
```
P->next = Q;  
Q->next = R;  
R->next = P;
```



(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 6: Doubly Linked Lists with Head and Tail (15 Points: 2/3/3/3/4)

Assume the declarations:

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

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 `Data`, `Next`, and `Back` and which uses these to set the internal fields `data`, `next`, and `back`. Arrange that each of the constructor parameters has a default (to zero, null, and null respectively).

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

(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, 0); // insert node at the head

    if (Tail == 0)
        Tail = Head; // adjust tail if first node
    else
        Head->next->back = Head; // adjust back link from old head
};
```

(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) {
    Tail = new Node(item, 0, Tail);    // insert node at the tail

    if (Head == 0)
        Head = Tail;                // adjust head if first node
    else
        Tail->back->next = Tail;     // adjust next link from old tail
};
```

(E) Write a member function `DeleteList` of class `List` that will delete each node in the list and then set the `Head` and `Tail` pointers so that the list is considered empty.

```
void DeleteList() {
    NodePointer P;

    while(Head) {
        P = Head;
        Head = head->next;
        delete P;
    }

    Tail = 0;
};
```