**COEN 79**

## Object-Oriented Programming and Advanced Data Structures

**Assignment #4 - Due: November 1st, 2018 – 11:59PM**

**Name:** **Date:**

- Number of questions: 10
- Points per question: 0.4
- Total: 4 points

---

1. Write a function to remove duplicates from a *forward linked list*. *Do not use a temporary buffer*. *The ordering of items is not important*.

```
1.  void list_remove_dups(node* head_ptr)
2.  // Postcondition: All the duplicates are removed from the linked list
3.  // Example: If the list contains 1,1,1,2, after running this function the list
4.  // contains 1,2
```

Answer:
The basic idea is to use two pointers: a *cursor* and a *runner*. When the cursor is pointing to a node *i*, we use the runner to check if any of the nodes after *i* is equal to *i*. If yes, then we remove that node. After a round is completed, we move the cursor and repeat the same process.

```
1.  void list_remove_dups(node * head_ptr)
2.  // Postcondition: All the duplicates are removed from the liked list
3.  // EXAMPLE: If the list contains 1,1,1,2, after running this function the list
4.  // contains 1,2
5.  {
6.      if (head_ptr == NULL)
7.          return;
8.
9.      node* cursor = head_ptr;
10.     while (cursor != NULL)
11.     {
12.         node* runner = cursor;
13.
14.         while (runner -> link() != NULL)
15.         {
16.             if (cursor -> data() == runner -> link() -> data())
17.             {
18.                 node* tmp_delete = runner -> link();
19.                 node* tmp_link = runner -> link() -> link();
20.                 runner -> set_link(tmp_link);
21.                 delete tmp_delete;
22.             }
23.             else
24.                 runner = runner -> link();
25.         }
26.     cursor = cursor - > link();
27.     }
28. }
```

2. The node class is defined as follows:

```
class node {
    public: // TYPEDEF
        typedef double value_type;

    // CONSTRUCTOR
    node(const value_type& init_data = value_type(), node* init_link = NULL) {
            data_field = init_data;
            link_field = init_link;  }

    // Member functions to set the data and link fields:
    void set_data(const value_type& new_data) { data_field = new_data; }
    void set_link(node* new_link) { link_field = new_link;  }

    // Constant member function to retrieve the current data:
    value_type data() const { return data_field; }

    // Two slightly different member functions to retreive the current link:
    const node* link() const { return link_field; }
    node* link() { return link_field; }

    private:
        value_type data_field;
        node* link_field;
};
```

Implement the following function. (Note: *No toolkit function is available*. Only the node class is available.)

```
void list_copy (const node* source_ptr, node*& head_ptr, node*& tail_ptr)
// Precondition: source_ptr is the head pointer of a linked list.
// Postcondition: head_ptr and tail_ptr are the head and tail pointers for a new list that
// contains the same items as the list pointed to by source_ptr
{
    head_ptr = NULL;
    tail_ptr = NULL;

    // Handle the case of the empty list.
    if (source_ptr == NULL) return;

    // Copy the head node first
    const node* cursor = source_ptr;
    head_ptr = tail_ptr = new node(cursor -> data(), NULL);
    cursor = cursor -> link();

    // Copy rest of the nodes
    while (cursor != NULL)
    {
        tail_ptr - > set_link( new node(cursor - > data(), NULL) );
        tail_ptr = tail_ptr - > link();
        cursor = cursor - > link();
    }
}
```

3. Please justify why the linked list toolkit functions are not member functions of the node class?

Answer:
The reason is that we need to be able to run these functions even if the linked list is empty. If we implement these functions as members of the node class, then if the linked list is empty, *there is no object to activate the function*.
For example, assume `list_length` has been implemented as a member function of node class. If the list is not empty, then we can activate the function as `head_ptr->list_lenght()`. However, if the list is empty, `head_ptr->list_lenght()` results in program crash because there is no object to activate the `list_length` function.

4. In the following function, why `cursor` has been declared as a `const` variable? What happens if you change it to non-const?

```
1.  size_t list_length (const node* head_ptr)
2.  // Precondition: head_ptr is the head pointer of a linked list.
3.  // Postcondition: The value returned is the number of nodes in the // linked list.
4.  {
5.          const node* cursor;
6.          size_t answer;
7.          answer = 0;
8.          for (cursor = head_ptr; cursor != NULL; cursor = cursor - > link())
9.              ++answer;
10.         return answer;
11. }
```

Answer:
`cursor`, is declared using the `const` keyword because we do not want to apply any changes to the linked list, and the `head_ptr` parameter is `const`. If `cursor` was not `const`, then the compiler would not permit the assignment `cursor = head_ptr`.

5. What is the output of this code? Please explain your answer.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class student {
5.  public:
6.      static int ctor;
7.      static int cc;
8.      static int dest;
9.      static int asop;
10.     student() {
11.         name = "scu";
12.         ++ctor;
13.     };
14.     student(const student & source) {
15.         this -> name = source.name;
16.         this -> id = source.id;
17.         ++cc;
18.     };
19.     ~student() {
20.         ++dest;
21.     };
22.     void operator = (const student & source) {
23.         this -> name = source.name;
24.         this -> id = source.id;
25.         ++asop;
26.     }
27. private:
28.     string name;
29.     int id;
30. };
31.
32. int student::ctor= 0;
33. int student::cc = 0;
34. int student::dest = 0;
35. int student::asop = 0;

36. student stuFunc(student input) {
37.     return input;
38. }
39.
40. int main(int argc, const char * argv[]) {
41.
42.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
           student::dest << "  asop = " << student::asop << endl;
43.
44.     student mySt1;
45.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                      student::dest << "  asop = " << student::asop << endl;
46.
47.     stuFunc(mySt1);
48.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                    student::dest << "  asop = " << student::asop << endl;
49.
50.     student mySt2 = stuFunc(mySt1);
51.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                    student::dest << "  asop = " << student::asop << endl;
52.
```

```
53.     return 0;
54. }
```

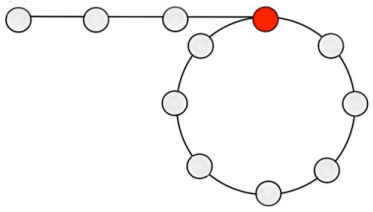
## Answer:

The output is:

```
Line 42: ctor = 0  cc = 0  dest = 0  asop = 0

After "student mySt1"
Line 45: ctor = 1  cc = 0  dest = 0  asop = 0

After "stuFunc(mySt1)"
Line 48: ctor = 1  cc = 2  dest = 2  asop = 0

After "student mySt2 = stuFunc(mySt1)"
Line 51: ctor = 1  cc = 4  dest = 3  asop = 0
```

- **Line 44**: we create an object of student, `mySt1`, on stack memory.
  - This results in calling the `student` constructor. Therefore, `ctor` is increased by 1.
- **Line 47**: we pass the object `mySt1` to `stuFunc` by value.
  - This results in calling the copy constructor for constructing the value parameter.
  - In addition, the return value of the function is constructed using the copy constructor. When the function returns, the local variable `input` is destructed.
  - Also, since the return value has not been assigned to any variable, the returned value is destructed.
  - Therefore, `dest` is increased by 2.
- **Line 50**: the copy constructor is called twice, once for the local variable, and once for returning the value and assigning it to the variable `mySt2`.
  - However, the destructor is called once because we assign the return value to the variable `mySt2` and therefore the returned value is not destructed and it is in fact, the newly created object `mySt2`.

6. Given a circular forward linked list, write an algorithm that returns a pointer to the node at the beginning of the loop.



```
1.  node* list_detect_loop(node* head_ptr) {
2.  // Pre-condition: head_ptr is the head pointer of the linked list
3.  // Post-condition: The return value is a pointer to the beginning of the loop
4.  // Returns NULL if no loop has been detected.
5.
6.      node* slow_runner = head_ptr;
7.      node* fast_runner = head_ptr;
8.
9.      // Find the meeting point. The meeting point is LOOP_SIZE - K steps inside the loop.
10.     while (fast_runner != NULL && fast_runner -> link() != NULL)
11.     {
12.
13.         // Move slow_runner one step at a time
14.         slow_runner = slow_runner -> link();
15.
16.         // Move slow_runner two steps at a time
17.         fast_runner = fast_runner -> link() -> link();
18.
19.         // Stop moving the runners when they meet
20.         if (slow_runner == fast_runner) {   break;  }
21.     }
22.
23.
24.     //Error check. No meeting point means no loop
25.     if (fast_runner == NULL || fast_runner -> link() == NULL)
26.         return NULL;
27.
28.     // Move slow_runner to the head, and keep fast_runner at the merging point.
29.     // Both pointers are k steps from the loop start.
30.     // We need this step because we don't know the value of k.
31.     slow_runner = head_ptr;
32.
33.     while (slow_runner != fast_runner) {
34.         slow_runner = slow_runner - > link();
35.         fast_runner = fast_runner - > link();
36.     }
37.
38.     return fast_runner;
39. }
```

7. Suppose `f` is a function with a prototype like this:

```
1. void f(________ head_ptr);
2. // Precondition: head_ptr is a head pointer for a linked list.
3. // Postcondition: The function f has done some manipulation of the linked list,
4. // and the list might now have a new head node.
```

What is the best data type for `head_ptr` in this function?

A. `node`
B. `node&`
C. `node*`
D. `node*&`

Answer: D

8. Why does our node class have two versions of the link member function?

A. One is public, the other is private.
B. One is to use with a const pointer, the other with a regular pointer.
C. One returns the forward link, the other returns the backward link.
D. One returns the data, the other returns a pointer to the next node.

Answer: B

9. Discuss the two major effects of storing elements of a data structure in contiguous memory locations. Hint: Discuss random access operator, and compare the speed of arrays and linked lists when used in real applications.

Answer:
Arrays store elements in contiguous memory locations. This enables us to access the elements using the random-access operator. For example, we can simply access the $n^{th}$ element of array `var` using notation `var[n]`.
However, to access the $n^{th}$ element of a linked list, we have to traverse the elements preceding to that node.

The other effect of storing elements in contiguous memory locations is enhanced performance through cache memory. In fact, processors use a cache memory, which is smaller and faster memory compared to the main memory (i.e., RAM). When you access $i^{th}$ element of an array, the processor transfers the data around the storage

location of `var[i]` to the cache memory. Therefore, the processor can use cache memory to perform operations on the array, instead of referring to RAM.
In contrast, as a linked list's nodes are not stored in contiguous memory locations, the chance of bringing the linked list nodes to the cache memory is lower. Therefore, the processor will need to continuously access main memory in order to operate on a linked list.
Although the complexity of insertion and deletion from an array and a linked list are O(n) and O(1), respectively, in reality, we may observe higher performance of array compared to linked list, depending on the hardware characteristics (i.e., processor and RAM).

10. The bag class is defined as follows:

```
1.  class bag {
2.      public:
3.      // TYPEDEFS
4.      typedef std::size_t size_type;
5.      typedef node::value_type value_type;
6.
7.      // CONSTRUCTORS and DESTRUCTOR
8.      bag();
9.      bag(const bag & source);
10.     ~bag();
11.
12.     // MODIFICATION MEMBER FUNCTIONS
13.     size_type erase(const value_type & target);
14.     bool erase_one(const value_type & target);
15.     void insert(const value_type & entry);
16.     void operator += (const bag & addend);
17.     void operator = (const bag & source);
18.
19.     // CONSTANT MEMBER FUNCTIONS
20.     size_type size() const {  return many_nodes;  }
21.     size_type count(const value_type & target) const;
22.     value_type grab() const;
23.
24.     private:
25.         node* head_ptr; // List head pointer
26.         size_type many_nodes; // Number of nodes on the list
27. };
```

Considering the `node` class definition given in Question 2, implement the assignment operator for the `bag` class. (Note: *No toolkit function is available.*)

```
1.  void bag::operator = (const bag & source)
2.  // Library facilities used: node1.h
3.  {
4.          if (this == & source) return;
5.
```

```
6.          // Delete the linked list
7.          node* cursor = head_ptr;
8.          while (cursor != NULL) {
9.              cursor = cursor -> link();
10.             delete head_ptr;
11.             head_ptr = cursor;
12.         }
13.
14.         many_nodes = 0;
15.
16.         // Copy the linked list of "source" object
17.         head_ptr = NULL;
18.         node * tail_ptr = NULL;
19.
20.         // Handle the case of the empty list.
21.         if (source.head_ptr == NULL)  return;
22.
23.         const node* runner = source.head_ptr;
24.         head_ptr = tail_ptr = new node(runner -> data(), NULL);
25.         runner = runner -> link();
26.
27.         while (runner != NULL) {
28.             tail_ptr -> set_link(new node(runner -> data(), NULL));
29.             tail_ptr = tail_ptr -> link();
30.             runner = runner -> link();
31.         }
32.
33.         many_nodes = source.many_nodes;
34.     }
```