Stephen Tambussi
5/13/2020
COEN 79

1)

```
void list_copy(const node* source_ptr, node*& head_ptr, node*& tail_ptr)
{
    assert(source_ptr!=NULL);
    head_ptr = NULL;
    tail_ptr = NULL;
    head_ptr = new node(source_ptr->data(), head_ptr);//allocates memory for first node
    tail_ptr = head_ptr;
    node* cursor = source_ptr->link();
    while(cursor != NULL)
    {
        node* insert = new node;
        insert->set_data(cursor->data());
        insert->set_link(tail_ptr->link());
        tail_ptr->set_link(insert);
        //tailptr's node now points at inserted node to list
        tail_ptr = tail_ptr->link();
        //tailptr now points at insert/inserted node becomes the tail
        cursor = cursor->link();
        //iterates through list
    }
}
```

2) The toolkit functions rely on an actual object of the node class existing(being declared and initialized) so they can't be member functions of the node class because they can't operate on an empty list.
3) Cursor has been declared as a const variable because it is not being used to modify the list by the function. If cursor was changed to a non-const variable it couldn't be assigned to head_ptr because the compiler won't assign a non-const variable to a const one.
4) The data structure to use for the mobile app example is a doubly linked list. A doubly linked list works best because each node in the list represents a screen on the mobile

app. When the back arrow is pressed, the previous screen is shown through the node's link(pointer) to the previous node. This concept applies to the forward arrow as well.

5) B-One is to use with a const pointer, the other with a regular pointer.
6) Element insert: all iterators and references unaffected
   Element Deletion: only the iterators and references of the erased element are invalidated
7) Element insert: all iterators and references before the point of insertion are unaffected. If the new container's size is greater than the previous capacity, then all iterators and references are invalidated
   Element Deletion: every iterator and reference after the point of deletion are invalidated
8) A random access iterator has all the abilities of bidirectional iterators: read/write and forward/backward moving. Random access iterators also have the ability to quickly access any arbitrary location in a container, hence the name. A vector and deque offer random access iterators.
9)

```
template < class Item >
class bag_iterator: public std::iterator < std::forward_iterator_tag, Item
>
{
    private:
        size_type capacity;
        size_type used;
        size_type current;
        Item* data;
    public:
        typedef std::size_t size_type;
        bag_iterator(size_type capacity, size_type used, size_type
current, Item* data)
        {
            this->capacity = capacity;
            this->used = used;
            this->current = current;
            this->data = data;
        }
        Item& operator* () const
        {
            return *data;
        }
        bag_iterator& operator++()//prefix
        {
            data++;
            return *this;
        }
        bag_iterator operator++(int)//postfix
        {
            bag_iterator og = *this;
            ++(*this);
            return og;
        }
        bool operator ==(const bag_iterator other) const
        {
            return(capacity == other.capacity && used == other.used
            && current == other.current &&
            for(int i = 0; i < used; i++)
            {
                data[i] == other.data[i];
            });
        }
        bool operator != (const bag_iterator other) const
        {
            return (capacity != other.capacity || used != other.used ||
            current != other.current || for(int i = 0; i < used; i++)
            {
                data[i] != other.data[i];
            });
        }
}
```