```
#include <iostream>

using namespace std;

int A[100]; // creates array of size 100

/**
 * @brief
 * Node item that stores a variable.
 * Overloaded to accept a custom variable 'x'
 */
struct Node {
    int x;
    Node *next;
    Node() {
        x = -1;
        next = NULL;
    }
    Node(int n) {
        x = n;
        next = NULL;
    }
};

/**
 * @brief Class List
 *
 * Private:
 * Node pointer 'Head' - points to front of list
 * Node pointer 'Tail' - points to rear of list
 *
 * Public:
 * List() -               creates empty list
 *
 * void Push(int) -       pushes a new node at the
 *                        end of the list. Size++
 *
 * void Insert(int) -     pushes a new node to front
 *                        of list. Size++
 *
 * void PrintTail() -     prints value in tail node
 *
 * void Print() -         traverses list and print node
 *                        values.
 *
 * int Pop() -            not implemented
 *
 * List operator+(List) - overloaded operator '+'
 *                        adds the values of both lists
 *                        and returns new list of new values
 *
 * int operator[](int) - overloaded brackets that allows user to
```

```
 *                      to traverse list like it's an array.
 *                      Returns value in node at given index.
 */
class List {
private:
    Node *Head;
    Node *Tail;
    int Size;

public:
    List() {
        Head = Tail = NULL;
        Size = 0;
    }

    void Push(int val) {
        // allocate new memory and init node
        Node *Temp = new Node(val);

        if (!Head && !Tail) {
            Head = Tail = Temp;
        } else {
            Tail->next = Temp;
            Tail = Temp;
        }
        Size++;
    }

    void Insert(int val) {
        // allocate new memory and init node
        Node *Temp = new Node(val);

        // figure out where it goes in the list

        Temp->next = Head;
        Head = Temp;
        if (!Tail) {
            Tail = Head;
        }
        Size++;
    }

    void PrintTail() {
        cout << Tail->x << endl;
    }

    string Print() {
        Node *Temp = Head;
        string list;

        while (Temp != NULL) {
            list += to_string(Temp->x) + "->";
            Temp = Temp->next;
        }
```

```
    return list;
}

// not implemented
int Pop() {
    Size--;
    return 0; //
}

List operator+(const List &Rhs) {
    // Create a new list that will contain both when done
    List NewList;

    // Get a reference to beginning of local list
    Node *Temp = Head;

    // Loop through local list and Push values onto new list
    while (Temp != NULL) {
        NewList.Push(Temp->x);
        Temp = Temp->next;
    }

    // Get a reference to head of Rhs
    Temp = Rhs.Head;

    // Same as above, loop and push
    while (Temp != NULL) {
        NewList.Push(Temp->x);
        Temp = Temp->next;
    }

    // Return new concatenated version of lists
    return NewList;
}

// Implementation of [] operator.  This function returns an
// int value as if the list were an array.
int operator[](int index) {
    Node *Temp = Head;

    if (index >= Size) {
        cout << "Index out of bounds, exiting";
        exit(0);
    } else {

        for (int i = 0; i < index; i++) {
            Temp = Temp->next;
        }
        return Temp->x;
    }
}

friend ostream &operator<<(ostream &os, List L) {
```

```
        os << L.Print();
        return os;
    }
};

int main(int argc, char **argv) {
    List L1;
    List L2;

    for (int i = 0; i < 25; i++) {
        L1.Push(i);
    }

    for (int i = 50; i < 100; i++) {
        L2.Push(i);
    }

    //cout << L1 << endl;
    L1.PrintTail();
    L2.PrintTail();

    List L3 = L1 + L2;
    cout << L3 << endl;

    cout << L3[5] << endl;
    return 0;
}
```