3. Assume you have the following function that creates and inserts a node given a value before the element at a specified position in the list:

   ```
   void InsertValue(LinkedList* list, DataType_t val, int position);
   ```

   Note that a position of 0 inserts it at the start. Draw the resulting doubly linked list in each of the following steps:

   ```
   LinkedList* list = new LinkedList;
   InsertValue(list, 1, 0);
   InsertValue(list, 2, 0);
   InsertValue(list, 3, 1);
   InsertValue(list, 4, 0);
   InsertValue(list, 5, 3);
   delete list;
   ```

2.6 Lab 1

The `int` Data type

Consider the int type and its modifiers: short, long, and unsigned. The size of the storage space, measured in bytes, for the different int types is implementation or machines-dependent. The range of values that a variable of a given int type can take depends on the allocated size and on whether it is signed (default)or unsigned. In this lab, we will write a program that uses the geometric sequence with common ratio r = 2 to cause an integer overflow. When the program runs, it reveals the actual sizes based on the underlying compiler and machine. The lab also demonstrates the issue of integer overflow that a programmer needs to be aware of. The code for the program is given below for the short int type. Experiment with it to explore the actual sizes and ranges of values for the different integer types using all possible combinations of modifiers. Note that you need to make changes to the number of terms `N_TERMS`.

```
/* Code uses geometric series with common ratio 2 to reveal the sizes
 * and ranges of different int types and to demonstrate the integer
 * overflow problem.
 */
```

```
#include <stdio.h>
#include <stdlib.h> // for using exit(EXIT_SUCCESS)instead of return 0
#include <iostream>
using namespace std;
#define N_TERMS 16 // number of term in the geometric sequence.

int main ()
{
    int seq_term, seq_sum;

    seq_term = seq_sum = 1;
    cout << "Term no. " << 1 << " = " << seq_term;
    cout << " and accumulated sum = " << seq_sum << "." << endl;
    for (int i = 1; i < N_TERMS; i++)
    {
        seq_term *= 2;  // double itself
        seq_sum += seq_term;  // add to tracking sum
        cout << "Term no." << i+1 << " = " << seq_term ;
        cout << " and accumulated sum = "<< seq_sum << ".\n";
    }
    system("pause");  // pause so we can view output
}
```

Pointers

Using the following program to test your understanding of pointers and of one of their uses.

```
/*Pointer variables and passing pointer parameters*/

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

// Declare functions that we will define later on (so we can use them in main()).
void bad_charSwap (char v1, char v2);
void good_charSwap (char *v1, char *v2);
void bad_charPtrSwap (char *p1, char *p2);
void good_charPtrSwap (char **p1, char **p2);
```

```
int main ()
{
    char c1,c2;
    char *cPtr1, *cPtr2;
    // At this point, cPtr1 and cPtr2 have arbitrary memory addresses.
    // They are considered "wild" or "bad". They should NOT be dereferenced
    // until they are assigned the address of an allocated storage block.
    // That is, one should not attempt to read from or write at the
    // addresses they contain.
    c1 = 'A';
    c2 = 'B';
    cPtr1 = &c1;
    cPtr2 = &c2;

    // At this point, cPtr1 and cPtr2 are assigned the addresses allocated
    // for c1 and c2, respectively. Hence, they can be dereferenced.

    cout << ">>> After the assignment statements:" << endl;
    cout << "c1 = " << c1 << ", c2 = " << c2 << endl;
    // Let's now read the pointers.
    // We static_cast from char* to void* because cout tries to interpret char*
    // as a null-terminated array of characters. But what we want to output is
    // the address of the pointer. Don't worry if you don't get this - just know
    // that we are doing this to output the memory address of the variable.
    cout << "cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
      << *cPtr1 << endl;
    cout << "cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
      << *cPtr2 << endl << endl;

    /* -------------------Swapping char values -----------------*/
    cout << ">>> Swapping char values ..." << endl;

    bad_charSwap(c1, c2);
    cout << ">>> After calling bad_charSwap(c1,c2):" << endl;
    cout << "c1 = " << c1 << ", c2 = " << c2 <<  endl;
    cout << "cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
         << *cPtr1 << endl;
    cout << "cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
         << *cPtr2 << endl << endl;
    good_charSwap(&c1, &c2);
    cout << ">>> After calling good_charSwap(&c1,&c2):" << endl;
```

```
    cout << "c1 = " << c1 << ", c2 = " << c2 <<  endl;
    cout << "cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
         << *cPtr1 << endl;
    cout << "cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
         << *cPtr2 << endl << endl;

    good_charSwap(cPtr1, cPtr2);
    cout << ">>> After calling good_charSwap(cPtr1, cPtr2):" << endl;
    cout << "c1 = " << *cPtr1 << ", c2 = " << *cPtr2 <<  endl;
    cout << "cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
         << *cPtr1 << endl;
    cout << "cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
         << *cPtr2 << endl << endl;

    /* -------------------Swapping Pointers to char  -----------------*/

    cout << ">>> Swapping pointers ..." << endl;

    bad_charPtrSwap(cPtr1, cPtr2);
    cout << ">>> After calling bad_charPtrSwap(cPtr1,cPtr2):" << endl;
    cout << "cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
         << *cPtr1 << endl;
    cout << "cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
         << *cPtr2 << endl << endl;

    good_charPtrSwap(&cPtr1, &cPtr2);
    cout <<">>> After calling good_charPtrSwap(&cPtr1,&cPtr2):" << endl;
    cout <<"cPtr1 = " << static_cast<void*>(cPtr1) << " and references value "
         << *cPtr1 << endl;
    cout <<"cPtr2 = " << static_cast<void*>(cPtr2) << " and references value "
         << *cPtr2 << endl << endl;

    system("pause");
}
void bad_charSwap(char v1, char v2)
{
    char temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
void good_charSwap(char *v1, char *v2)
```

```
{
    char temp;
    temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
void bad_charPtrSwap(char *p1, char *p2)
{
    char *temp;
    temp = p1;
    p1 = p2;
    p2 = temp;
}
void good_charPtrSwap(char **p1, char **p2)
{
    char *temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Basic Linked List Operations

Consider the source file shown below (delimited by the label "linked_list.cpp"), which contains the type definitions and function prototypes. Complete the code as follows:

- Implement the body of each function.
- After implementing a function, debug it and test it by making calls from the main function, as shown by the examples in the main given further below. Make other testing examples.
- Compare your implementation with the code given in Sec.3, make any necessary fixes. Debug and test again with examples.
- Optional: when the functions have been tested, replace the hard-coded calls in the main function by writing code for a command-line interface. The interface would take a command symbol from the user as follows: i(InsertNewLast), d(DeleteLastNode), s(ListSearch), p(PrintList) or q(quit). In the case of i and s, the user should enter an integer data value. Debug and test the interface code separately. When it is working use it to make corresponding function calls and test the program.