

Introduction

In this assignment you'll build a unidirectional ("one-way") linked list. A linked list is a common data structure that is a collection of nodes which together represent a sequence. In a unidirectional linked list, each node points to the next node in the sequence.

```
class Node {  
public:  
    Node(int iData) : m_iData(iData), mp_nodeNext(nullptr) { }  
    ~Node() { }  
public:  
    // These are declared public so that they can be accessed  
    // without using member accessor functions.  
    int m_iData;  
    Node* mp_nodeNext;
```

In this project, each node in the linked list is represented by a Node class whose member variables include a data element, `m_iData`, which holds the data associated with the node, and a pointer, `mp_nodeNext`, which points to the next Node object in the list.

You'll also implement a function that displays the linked list. The function will be a static member function of the Node class called `displayList`.

```

class Node {
public:
    Node(int iData) : m_iData(iData), mp_nodeNext(nullptr) { }
    ~Node() { }
public:
    // These are declared public so that they can be accessed
    // without using member accessor functions.
    int m_iData;
    Node* mp_nodeNext;

public:
    /// Display a list of Node objects to standard output given
    /// the head of the list of Nodes. (This is a static
    /// member function; it behaves differently from ordinary
    /// member functions. You cannot use this->m_iData within
    /// this fuction, for example.)
    /// @param [in] p_nodeHead pointer to Node object that is the
    /// head node of a list of Node objects.
    /// @return void
    static void displayList(Node* p_nodeHead)

```

The main function that builds the linked list and calls displayList() is provided, along with a partial implementation of the Node class, in source file LinkedList_incomplete.cpp, downloadable from Canvas.

The main function declares three pointers to Node objects and assigns them to three Node objects allocated within heap memory ("allocated on the heap"): p_node1, p_node2 and p_node3. A pointer to the head of the linked list is then declared, p_nodeHead, and this is set to point to p_node1, which points to the first Node object in the linked list, called the head node. (When a pointer is said to point to a Node object, it implies the pointer contains the address of the start of the Node object, which resides in heap memory.)

```

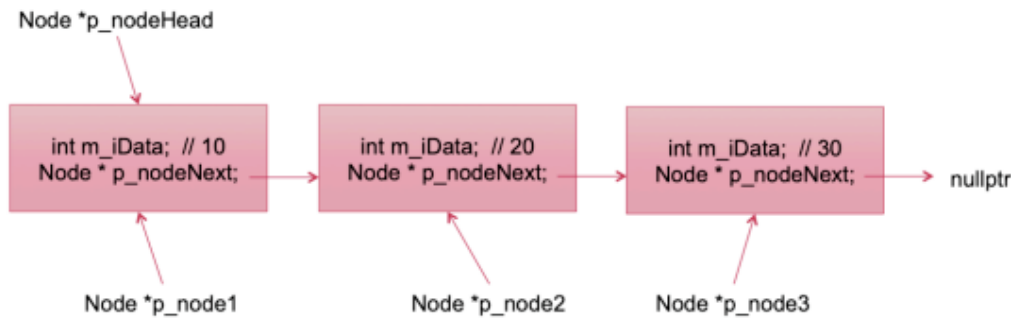
int main()
{
    // Allocate three node objects within heap memory.
    Node* p_node1 = new Node(10);
    Node* p_node2 = new Node(20);
    Node* p_node3 = new Node(30);

    // Set p_nodeHead to point to the head or leading node
    // in the list of nodes, in this case p_node1.
    Node* p_nodeHead = p_node1;
    p_node1->mp_nodeNext = p_node2; // append p_node2 to the list of Node objects after p_node1.
    p_node2->mp_nodeNext = p_node3; // append p_node3 to the list of Node objects after p_node2.
    p_node3->mp_nodeNext = nullptr; // The end of the list is indicated by a mp_nodeNext value of nullptr.

    Node::displayList(p_nodeHead); // display the list of nodes to standard output.
}

```

The linked list is then built by setting the `mp_nodeNext` member variable to point to the next node in the list. For example: `p_node1->mp_nodeNext = p_node2`. (The next node after the node pointed to by `p_node1` is the node pointed to by `p_node2`.) The figure below shows the linked list that is built by the main program in diagrammatic form.



In the second half of the assignment, you'll convert the `Node` class to a class template. A class template is a family of classes. It allows the functionality of the class to be adapted to different data types without repeating the entire code for each type. This will allow the class to store any type of data, not just an `int`.

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

The Assignment

Download the `LinkedList_incomplete.cpp` C++ source file from Canvas. The assignment is performed in two parts. In part 1, you'll complete the Link List program. In part 2, you'll modify the program to use templates.

Part 1

Edit the file to complete the `displayList` static member function.

```

/// Display a list of Node objects to standard output given
/// the head of the list of Nodes. (This is a static
/// member function; it behaves differently from ordinary
/// member functions. You cannot use this->m_iData within
/// this fuction, for example.)
/// @param [in] p_nodeHead pointer to Node object that is the
/// head node of a list of Node objects.
/// @return void
static void displayList(Node* p_nodeHead)

```

```

{
    Node* p_node = p_nodeHead;
    // Iterate over the list of Nodes beginning from the
    // head of the list. Print the data out for each node.
    // Then, set p_node to point to the next node in the list.
    // Continue until mp_nodeNext is set to nullptr, indicating
    // the end of the list.
    //
    // TODO: Insert a while loop that continues to iterate
    //       so long as the p_node pointer is not the null
    //       pointer, nullptr.

    {
        // TODO: Insert cout statement to display the data.

        // TODO: Update p_node to point to the next node in
        //       the linked list.
    }

    return;
}

```

The local variable `p_node` is initialized to `p_nodeHead`, the head node of the linked list.

Step 1A

First, insert a while statement after the first TODO within `displayList` that will continue to iterate so long as the `p_node` local variable is not equal to the null pointer, `nullptr`.

```
while (p_node != nullptr)
```

`p_node` will need to be updated with the loop, otherwise the loop will iterate forever.

Step 1B

Before we address that, let's print out the value of the data in the Node object. Insert the following `cout` statement after the second TODO in the `displayList()` static member function.

```
std::cout << p_node->m_iData << std::endl;
```

The `m_iData` member variable is public and so can be accessed from `p_node` directly without needing to use a member access function. This approach is used as a convenience here; accessor methods (such as `int getData()`) are preferred, though.

Step 1C

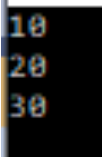
Next, the pointer `p_node` must be advanced to point to the next node in the linked list. Insert the following statement after the third TODO in the `displayList` static member function.

```
p_node = p_node->mp_nodeNext;
```

Step 1D

Build and execute your program.

The figure below shows how the data in each of the three Node objects in the linked list should be displayed.



```
10
20
30
```

You now have a linked list program working for a Node that supports integer types. But what if you want to support type double instead of int. For that, you need to recast the Node class as a class template.

Part 2

In part 2 you'll modify your linked list class to convert Node to a class template and then modify the main() function to use the new class template.

Step 2A

First, you'll convert the Node class to a class template. Make the following six changes to the Node class. Each change is identified by a TODO comment in the source code listing below. The listing shows how the changes should look.

```

template<typename T>                                     // TODO: Add the template delcaration
class Node {
public:
    Node(T iData) : m_iData(iData), mp_nodeNext(nullptr) { }    // TODO: change int to T in the Node constructor
    ~Node() { }
public:
    // These are declared public so that they can be accessed
    // without using member accessor functions.
    T m_iData;                                                    // TODO: Declared m_iData to be of type T
    Node<T>* mp_nodeNext;                                         // TODO: Change Node* to Node<T>*

public:
    /// Display a list of Node objects to standard output given
    /// the head of the list of Nodes. (This is a static
    /// member function; it behaves differently from ordinary
    /// member functions. You cannot use this->m_iData within
    /// this fuction, for example.)
    /// @param [in] p_nodeHead pointer to Node object that is the
    /// head node of a list of Node objects.
    /// @return void
    static void displayList(Node<T>* p_nodeHead)                // TODO: Change Node* to Node<T>*
    {
        Node<T>* p_node = p_nodeHead;                          // TODO: Change Node* to Node<T>*
        // Iterate over the list of Nodes beginning from the
        // head of the list. Print the data out for each node.
        // Then, set p_node to point to the next node in the list.
        // Continue until mp_nodeNext is set to nullptr, indicating
        // the end of the list.
        //
        // TODO: Insert a while loop that continues to iterate
        //       so long as the p_node pointer is not the null
        //       pointer, nullptr.
        while (p_node != nullptr)
        {
            // TODO: Insert cout statement to display the data.
            std::cout << p_node->m_iData << std::endl;
            // TODO: Update p_node to point to the next node in
            //       the liked list.
            p_node = p_node->mp_nodeNext;
        }
        return;
    }
};

```

Step 2B

Next, you'll modify the main function to use the new class template. Replace the existing code in main with the following. (The code listing below can be copied and pasted over the existing code in the main() function.)

```

// Allocate three node objects within heap memory.
Node<int>* p_node1 = new Node<int>(10);
Node<int>* p_node2 = new Node<int>(20);
Node<int>* p_node3 = new Node<int>(30);

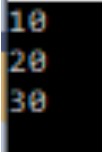
// Set p_nodeHead to point to the head or leading node
// in the list of nodes, in this case p_node1.
Node<int>* p_nodeHead = p_node1;
p_node1->mp_nodeNext = p_node2; // append p_node2 to the list of Node objects after p_node1.
p_node2->mp_nodeNext = p_node3; // append p_node3 to the list of Node objects after p_node2.
p_node3->mp_nodeNext = nullptr; // The end of the list is indicated by a mp_nodeNext value of
nullptr.

Node<int>::displayList(p_nodeHead); // display the list of nodes to standard output.

```

Step 2C

Compile and run your program. The output should be identical to that previously obtained.



```
10
20
30
```

Step 2D

Modify the main function again. Copy the existing code and paste it in to a new basic block. Surround the original code within its own basic block. (This allows the declarations to be repeated within two different scopes to avoid compiler errors due to repeated declarations.) In the second block of code, change the `<int>` to `<double>` as shown in the sample code below.

```
int main()
{
    {
        // Allocate three node objects within heap memory.
        Node<int>* p_node1 = new Node<int>(10);
        Node<int>* p_node2 = new Node<int>(20);
        Node<int>* p_node3 = new Node<int>(30);

        // Set p_nodeHead to point to the head or leading node
        // in the list of nodes, in this case p_node1.
        Node<int>* p_nodeHead = p_node1;
        p_node1->mp_nodeNext = p_node2; // append p_node2 to the list of Node objects after p_node1.
        p_node2->mp_nodeNext = p_node3; // append p_node3 to the list of Node objects after p_node2.
        p_node3->mp_nodeNext = nullptr; // The end of the list is indicated by a mp_nodeNext value of nullptr.

        Node<int>::displayList(p_nodeHead); // display the list of nodes to standard output.
    }

    {
        // Allocate three node objects within heap memory.
        Node<double>* p_node1 = new Node<double>(10.1);
        Node<double>* p_node2 = new Node<double>(20.2);
        Node<double>* p_node3 = new Node<double>(30.3);

        // Set p_nodeIHead to point to the head or leading node
        // in the list of nodes, in this case p_node1.
        Node<double>* p_nodeHead = p_node1;
        p_node1->mp_nodeNext = p_node2; // append p_node2 to the list of Node objects after p_node1.
        p_node2->mp_nodeNext = p_node3; // append p_node3 to the list of Node objects after p_node2.
        p_node3->mp_nodeNext = nullptr; // The end of the list is indicated by a mp_nodeNext value of nullptr.

        Node<double>::displayList(p_nodeHead); // display the list of nodes to standard output.
    }

    return 0;
}
```

Change the arguments to the Node constructors in the second block to literal constants of type double: 10.1, 20.2 and 30.3.

Step 2E

Build and run the program.

The output should appear as follows:

```
10  
20  
30  
10.1  
20.2  
30.3
```

You've now built a linked list class template that can support a variety of data types.