# 8.5 A first linked list

A common use of pointers is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for a vector. The following program illustrates how such a list can be created. A class is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list item, in this case just one int, and a pointer to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 8.5.1: A basic example to introduce linked lists.

```
#include <iostream>
using namespace std;

class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextLoc = nullptr);
   void InsertAfter(IntNode* nodePtr);
   IntNode* GetNext();
   void PrintNodeData();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
   this->dataVal = dataInit;
   this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = nullptr;

   tmpNext = this->nextNodePtr;    // Remember next
   this->nextNodePtr = nodeLoc;    // this -- node -- ?
   nodeLoc->nextNodePtr = tmpNext; // this -- node -- next
}

// Print dataVal
void IntNode::PrintNodeData() {
   cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
   return this->nextNodePtr;
}

int main() {
```

```
-1
555
777
999
```

```
int main() {
   IntNode* headObj  = nullptr; // Create IntNode objects
   IntNode* nodeObj1 = nullptr;
   IntNode* nodeObj2 = nullptr;
   IntNode* nodeObj3 = nullptr;
   IntNode* currObj  = nullptr;

   // Front of nodes list
   headObj = new IntNode(-1);

   // Insert nodes
   nodeObj1 = new IntNode(555);
   headObj->InsertAfter(nodeObj1);

   nodeObj2 = new IntNode(999);
   nodeObj1->InsertAfter(nodeObj2);

   nodeObj3 = new IntNode(777);
   nodeObj1->InsertAfter(nodeObj3);

   // Print linked list
   currObj = headObj;
   while (currObj != nullptr) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

Feedback?

**PARTICIPATION ACTIVITY** 8.5.1: Inserting nodes into a basic linked list.

1 2 3 4 5 2x speed

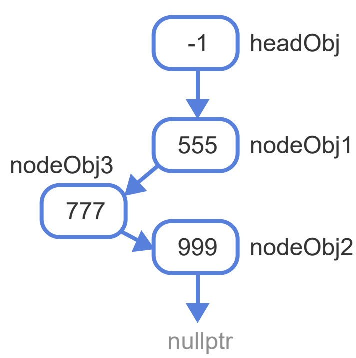
```
headObj = new IntNode(-1);

// Add nodeObj1 after headObj
nodeObj1 = new IntNode(555);
headObj->InsertAfter(nodeObj1);

...

// Add nodeObj3 after nodeObj1
nodeObj1->InsertAfter(nodeObj3);
```

```
tmpNext = this->nextNodePtr;
this->nextNodePtr = nodeLoc;
nodeLoc->nextNodePtr = tmpNext;
```




| | | |
|---|---|---|
| 75 | 86 | headObj |
| 76 | 84 | nodeObj1 |
| 77 | 82 | nodeObj2 |
| 78 | 80 | nodeObj3 |
| 79 | | |
| 80 | 777 | dataVal |
| 81 | 82 | nextNodePtr |
| 82 | 999 | dataVal |
| 83 | nullptr | nextNodePtr |
| 84 | 555 | dataVal |
| 85 | 80 | nextNodePtr |
| 86 | -1 | dataVal |
| 87 | 84 | nextNodePtr |

To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeOb
nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext

Feedback?

The most interesting part of the above program is the InsertAfter() function, which inserts a new node after a given node already in the list. The above animation illustrates.

**PARTICIPATION ACTIVITY** 8.5.2: A first linked list.

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or vector?

- (selected) An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
- Uses less memory overall.
- Can store items other than int variables.

**Correct**

Inserting only requires a couple of pointer updates.

2) What is the purpose of a list's head node?

- Stores the first item in the list.
- (selected) Provides a pointer to the first item's node in the list, if such an item exists.
- Stores all the data of the list.

**Correct**

The head points to the first item's node, or points to nothing if the list is empty.

3) After the above list is done having items inserted, at what memory address is the last list item's node located?

- 80
- 82
- 84
- 86

**Correct**

The last item is node2, which was allocated at address 82.

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

- Changes from 84 to 86.
- Changes from 84 to 82.
- Stays at 84.

**Correct**

The node does not have to be moved; only a few pointer values change.

Feedback?

In contrast to the above program that declares one variable for each item allocated by the new operator, a program commonly declares just one or a few variables to manage a large number of items allocated using the new operator. The following example replaces the above main() function, showing how just two pointer variables, currObj and lastObj, can manage 20 allocated items in the list.

To run the following figure, `#include <cstdlib>` was added to access the rand() function.

## Figure 8.5.2: Managing many new items using just a few pointer variables.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextLoc = nullptr);
   void InsertAfter(IntNode* nodePtr);
   IntNode* GetNext();
   void PrintNodeData();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nextLoc) {
   this->dataVal = dataInit;
   this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = nullptr;

   tmpNext = this->nextNodePtr;    // Remember next
   this->nextNodePtr = nodeLoc;    // this -- node -- ?
   nodeLoc->nextNodePtr = tmpNext; // this -- node -- next
}

// Print dataVal
void IntNode::PrintNodeData() {
   cout << this->dataVal << endl;
}

// Grab location pointed by nextNodePtr
IntNode* IntNode::GetNext() {
   return this->nextNodePtr;
}

int main() {
   IntNode* headObj = nullptr; // Create intNode objects
   IntNode* currObj = nullptr;
   IntNode* lastObj = nullptr;
   int i;                  // Loop index

   headObj = new IntNode(-1);        // Front of nodes list
   lastObj = headObj;

   for (i = 0; i < 20; ++i) {        // Append 20 rand nums
      currObj = new IntNode(rand());

      lastObj->InsertAfter(currObj); // Append curr
      lastObj = currObj;             // Curr is the new last item
   }

   currObj = headObj;                // Print the list

   while (currObj != nullptr) {
      currObj->PrintNodeData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

```
-1
16807
282475249
1622650073
984943658
1144108930
470211272
101027544
1457850878
1458777923
2007237709
823564440
1115438165
1784484492
74243042
114807987
1137522503
1441282327
16531729
823378840
143542612
```

Feedback?

## zyDE 8.5.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Load default template...

Run

```
class IntNode {
public:
   IntNode(int dataInit = 0, IntNode* nextL
   void InsertAfter(IntNode* nodePtr);
   IntNode* GetNext();
   void PrintNodeData();
   int GetDataVal();
private:
   int dataVal;
   IntNode* nextNodePtr;
};

// Constructor
IntNode::IntNode(int dataInit, IntNode* nex
   this->dataVal = dataInit;
   this->nextNodePtr = nextLoc;
}

/* Insert node after this node.
 * Before: this -- next
 * After:  this -- node -- next
 */
```

Feedback?

Normally, a linked list would be maintained by member functions of another class, such as IntList. Private data members of that class might include the list head (a list node allocated by the list class constructor), the list size, and the list tail (the last node in the list). Public member functions might include InsertAfter (insert a new node after the given node), PushBack (insert a new node after the last node), PushFront (insert a new node at the front of the list, just after the head), DeleteNode (deletes the node from the list), etc.

Exploring further:

- More on Linked Lists from cplusplus.com

**CHALLENGE ACTIVITY** 8.5.1: Enter the output of the program using Linked List.

Jump to level 1

Type the program's output.

```
-1
2
1
3
4
```

```
#include <iostream>
using namespace std;

class IntNode {
   public:
      IntNode(int value = -1, IntNode* nextLoc = nullptr);
      void InsertAfter(IntNode* nodePtr);
      int GetValue();
      IntNode* GetNext();
      void PrintData();
   private:
      int value;
      IntNode* nextIntNodePtr;
};

IntNode::IntNode(int val, IntNode* nextLoc) {
   this->value = val;
   this->nextIntNodePtr = nextLoc;
}

void IntNode::InsertAfter(IntNode* nodeLoc) {
   IntNode* tmpNext = nullptr;

   tmpNext = this->nextIntNodePtr;
   this->nextIntNodePtr = nodeLoc;
   nodeLoc->nextIntNodePtr = tmpNext;
}

IntNode* IntNode::GetNext() {
   return this->nextIntNodePtr;
}

void IntNode::PrintData() {
   cout << this->value << endl;
}

int main() {
   IntNode* headObj = nullptr;
   IntNode* node1 = nullptr;
   IntNode* node2 = nullptr;
   IntNode* node3 = nullptr;
   IntNode* node4 = nullptr;
   IntNode* currObj = nullptr;

   headObj = new IntNode(-1);

   node1 = new IntNode(1);
   headObj->InsertAfter(node1);

   node2 = new IntNode(2);
   headObj->InsertAfter(node2);

   node3 = new IntNode(3);
   node1->InsertAfter(node3);

   node4 = new IntNode(4);
   node3->InsertAfter(node4);

   currObj = headObj;

   while (currObj != nullptr) {
      currObj->PrintData();
      currObj = currObj->GetNext();
   }

   return 0;
}
```

1 2

Check Next **Done**. Click any level to practice more. Completion is preserv

✔ Following nodes after headObj node are inserted after the according node via InsertAfter(). order of the linked list is printed.

Yours

```
-1
2
1
3
4
```

Expected

```
-1
2
1
3
4
```

**Feedback?**

**CHALLENGE ACTIVITY** 8.5.2: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list.

```
      lastObj->InsertAfter(currObj); // Append curr
      lastObj = currObj;             // Curr is the new last item
   }

   currObj = headObj;                // Print the list
   while (currObj != nullptr) {
      cout << currObj->GetDataVal() << ", ";
      currObj = currObj->GetNext();
   }
   cout << endl;

   currObj = headObj;                // Count number of negative numbers
   while (currObj != nullptr) {

      /* Your solution goes here  */
      if(currObj->GetDataVal()<0){
         negativeCntr++;
      }


      currObj = currObj->GetNext();
   }
```

Run ✔ All tests passed

✓ Testing with 6 negatives

Your output
```
Number of negatives: 6
```

✓ Testing with 11 negatives

Your output
```
Number of negatives: 11
```

Feedback?