Chapter 14 | Linked Lists

CS185

Introduction to Linked Lists

Array Behavior

Arrays are simple, popular, and built-in to the C/C++ language and they have certain characteristics, both good and bad.

The Good:

- Built-in to the language.
- Easy to create at compile time or runtime.
- Accessing any element in the array is trivial (just use an index).
- Easy to "clean up". (Static arrays, just forget about it. Dynamic, use `delete` [])

The Bad:

- You need to know size ahead of time (both static and dynamic arrays).
- You must allocate a fixed amount of space (can't resize an array).
- Inserting and deleting anywhere but at the end requires a lot of work.

The Ugly:

- No bounds checking, allowing you to overwrite memory. (Most students are painfully aware of this by now.)
- The *unusual* pointer-array relationship here. (Look in the *Critique* section or search on "historical accidents or mistakes")

Example of the limitation of arrays: Reading integers from user into an array.

This is our algorithm:

1. Allocate an array to hold the numbers
2. Ask the user if they want to input a number
3. While the user wants to input a number
    a. Read in a number
    b. Add it to the end of the array
    c. Ask the user if they want to input another number

```
// Prints each value in the integer array
void print_array(int array[], int size)
{
  int i;
  for (i = 0; i < size; i++)
  {
    std::cout << array[i] << " ";
  }
  std::cout << std::endl;
}

int main(void)
{
  int numbers[30]; // Holds at most 30 integers
  int count = 0;
  char userDecision = 0;

  std::cout << "Would you like to input a number? (y/n)" << std::endl;
  std::cin >> userDecision;

  // As long as the user wants to enter numbers
  while (userDecision == 'y')
  {
    int number;

    std::cout << "Enter the number: ";
    std::cin >> number;
    // Add the number to the end of the array
    numbers[count++] = number;

    std::cout << "Would you like to input another number? (y/n)" << std::endl;
    std::cin >> userDecision;
  }

  // Print the array
  print_array(numbers, count);

  return 0;
}
```

Of course, if there are more than 30 numbers, we are going to overwrite the end of the array.

Possible "fixes":

- Don't read in more than 30 numbers.
- Set the size of the array to more than 30 (and hope it's big enough or waste a lot of space)
- Allocate the array at runtime. (Still need a size.) Choices:
  1. Ask the user for the size and allocate the array when the size is known.
  2. When the array is full, allocate another, bigger array, and copy old values into it. This may need to be done many times.

Linked Lists

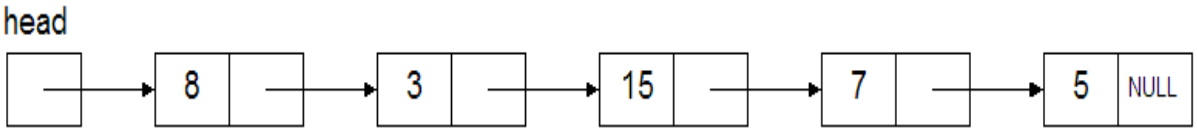
We'd like to overcome the limitations of arrays. One way is to use a *linked list*. So, what is a Linked List?

- A *dynamic* collection of elements called *nodes*.
- A node is a **struct** or **class** in C++.
- A node has two main portions
  - data portion -- same type (size) of information in all nodes
  - pointer portion -- a pointer to the next node in the list

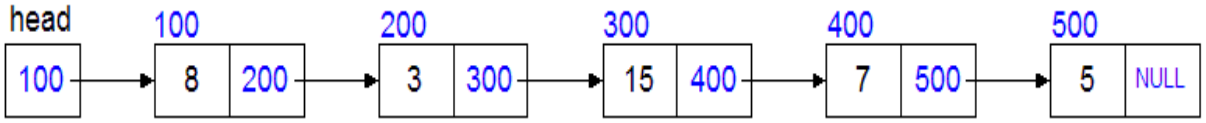
- The data portion can be as simple as an **int** or very complex. (It's user-defined.)
- The linked list is accessed through an external pointer called the *head* which points to the first node in the list.
- The last pointer in the list points to 0 (NULL), marking the end of the list. (Think of it as a "NULL-terminated" list)
- An example of a node structure (for a singly linked list) that contains an integer as it's data:

```
struct Node
{
  int number;          // data portion
  Node *next;        // pointer portion
};
```

- Example showing data as an integer, each node is 8 bytes (assume 32-bit pointers):



Example showing data as an integer, each node is 8bytes (assume 32-bit pointers):



- If the list is empty, head points to 0.
- If each node in the list has only one pointer (called a "next" pointer), the list is called a *singly linked list*
- Some lists have nodes with two pointers (called "next" and "previous"). This type of linked list is known as a *doubly linked list*.

Notice that the structure above is sort of *recursive*. In other words, we're defining the structure by including a reference to itself in the definition. (Actually, there's only a pointer to a structure of the same type.)

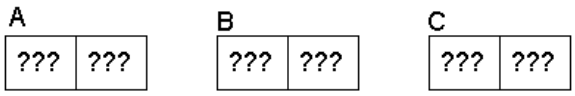
When the compiler encounters a structure member, it must know the size of the member. Since the size of all pointers is known at compile time, the code above is completely sane and legal. (Also, the compiler already knows what a `Node` is.)

This example code:

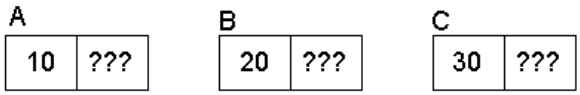
```
// #1 Declare 3 structs
Node A, B, C;

// #2 Set the 'data' portions of the nodes
A.number = 10;
B.number = 20;
C.number = 30;

// #3 Connect (link) the nodes together
A.next = &B;   // A's next points to B
B.next = &C;   // B's next points to C
C.next = 0;    // Nothing follows C
```

could be visualized as this:

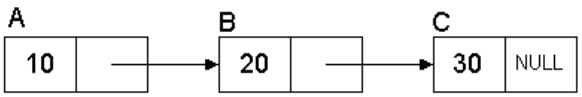
After #1



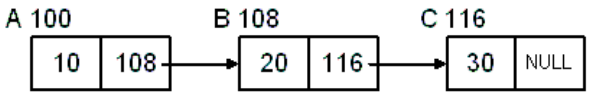
After #2



After #3



With arbitrary addresses



The "problem" with this approach, is that we are declaring (and naming) all of the nodes at compile time. If we wanted to read a list of 30 integers from a file, we'd need to declare 30 `Node` structs. We're worse off than with arrays.

Notice from the diagram that naming struct `B` and `C` is redundant. Also remember that we don't "name" our individual elements of an array. We refer to them by supplying a subscript on the array name:

```
int numbers[30]; // 30 "anonymous" elements
numbers[5] = 0;  // We don't have a "name" for the 6th element
```

This principle of "anonymous" elements will apply to linked lists as well:

- To access an element of an array, we simply use the name of the array (essentially a pointer to the first element) and an index (offset from the beginning).
- To access an element of a linked list, we use a pointer to the **first** node and then *walk* the list to find a particular node.

For example, with named nodes (as in the example above) we can print out the data of each node very simply:

```
std::cout << A.number << std::endl; // 10
std::cout << B.number << std::endl; // 20
std::cout << C.number << std::endl; // 30
```

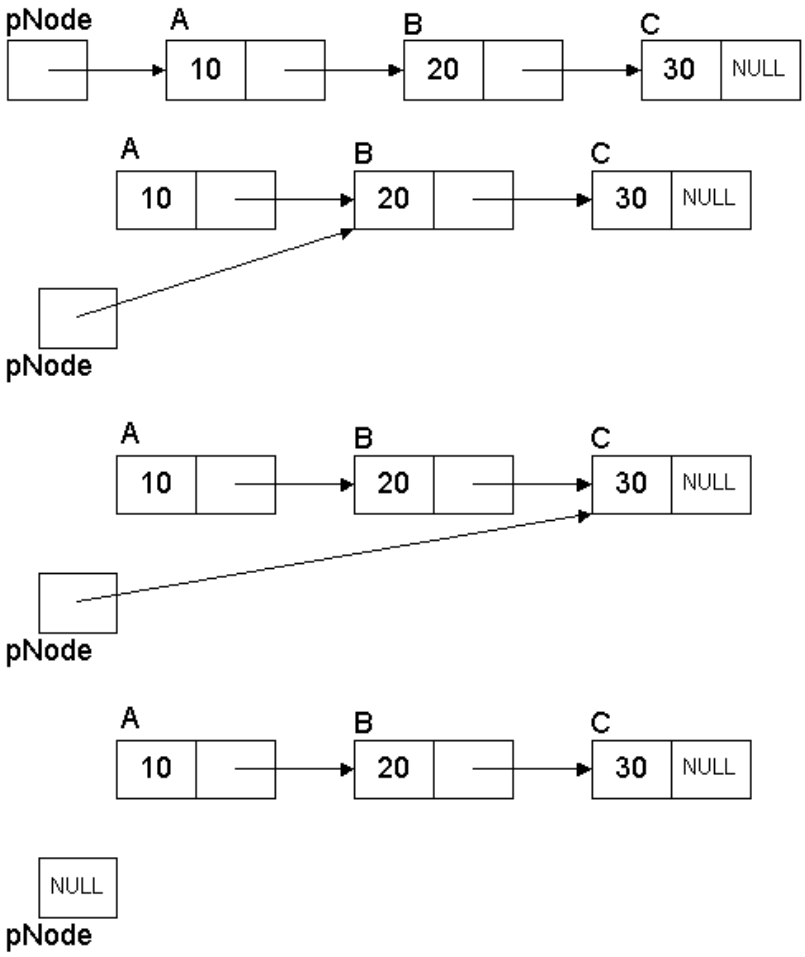
With unnamed nodes (i.e. access to the first node only):

```
std::cout << A.number << std::endl; // 10
std::cout << A.next->number << std::endl; // 20
std::cout << A.next->next->number << std::endl; // 30
```

Or, the more common method of using a loop to *"walk"* the list:

```
Node *pNode = &A; // Point to first node
while (pNode != NULL)
{
	std::cout << pNode->number << std::endl; // Print data
	pNode = pNode->next; // "Follow" the next pointer
}
```

Visually:



Problem Revisited

Let's revisit the original problem of storing an unknown number of integers from a user:

Linked List	Array
1. Ask the user if they want to input a number 2. While the user wants to input a number a. Read in a number b. Allocate a node to hold the number c. Add the node to the end of the list d. Ask the user if they want to input another number	1. Allocate an array to hold the numbers 2. Ask the user if they want to input a number 3. While the user wants to input a number a. Read in a number b. Add it to the end of the array c. Ask the user if they want to input another number

```
int main(void)
{
  Node *pList = 0; // empty list
  char userDecision = 0;

  std::cout << "Would you like to input a number? (y/n)" << std::endl;
  std::cin >> userDecision;

  while (userDecision == 'y')
  {
    Node *pNode;
    int number;

    // A. Get the next user number
    std::cout << "Enter the number: ";
    std::cin >> number;

    // B. Allocate a new node struct (same for all nodes)
    pNode = new Node;
    pNode->number = number; // Set the number
    pNode->next = 0;        // Set next (no next yet)

    // C. Add the new node to the end of the list
    // If the list is NULL (empty), this is the first
    // node we are adding to the list.
    if (pList == NULL)
    {
      pList = pNode;
    }
    else
    {
      // Find the end of the list
      Node *temp = pList;
      while (temp->next)
      {
        temp = temp->next;
      }

      temp->next = pNode; // Put new node at the end
    }

    std::cout << "Would you like to input another number? (y/n)" << std::endl;
    std::cin >> userDecision;
  }

  // Print the list
  print_list(pList);

  return 0;
}
```

Make sure you can follow what each line of code is doing. **You should definitely draw diagrams until you are very comfortable with linked lists.** (Which won't be until after you graduate, and even then you should still draw diagrams!)

Note these two sections especially:

Creating a new node for each element of data (number from the user):

```
// B. Allocate a new node struct (same for all nodes)
pNode = new Node;
pNode->number = number; // Set the number
pNode->next = 0;        // Set next (no next yet)
```

Adding the new node to the end of the list:

```
// C. Add the new node to the end of the list
// If the list is NULL (empty), this is the first
// node we are adding to the list.
if (pList == NULL)
{
  pList = pNode;
}
else
{
  // Find the end of the list
  Node *temp = pList;
  while (temp->next)
  {
    temp = temp->next;
  }

  temp->next = pNode; // Put new node at the end
}
```

Also note the `print_list` function used above:

```
void print_list(Node *list)
{
	while (list)
	{
		std::cout << list->number << " ";
		list = list->next;
	}
	std::cout << std::endl;
}
```

A few points to make so far:

- The code is certainly more complex than arrays, although not hard to understand.
- The number of nodes in a linked list is only dependent on the amount of memory in the computer. (This code can handle small lists or large lists.)
- We are only allocating what we need. (Arrays can waste space.)
- There is a 4-byte (size of a pointer on 32-bit computers) overhead for each node.
- The time it takes to add a node to the end of the linked list takes longer as the list grows.

- We must also remember to ***deallocate*** (delete) each node in the list when we are finished. (We haven't done that in the example yet.)

Also note that this very simple example does not do any error handling, especially the condition where `new` fails. In Real World™ code, you would need to check the result of `new` and deal with it accordingly.

Adding Nodes

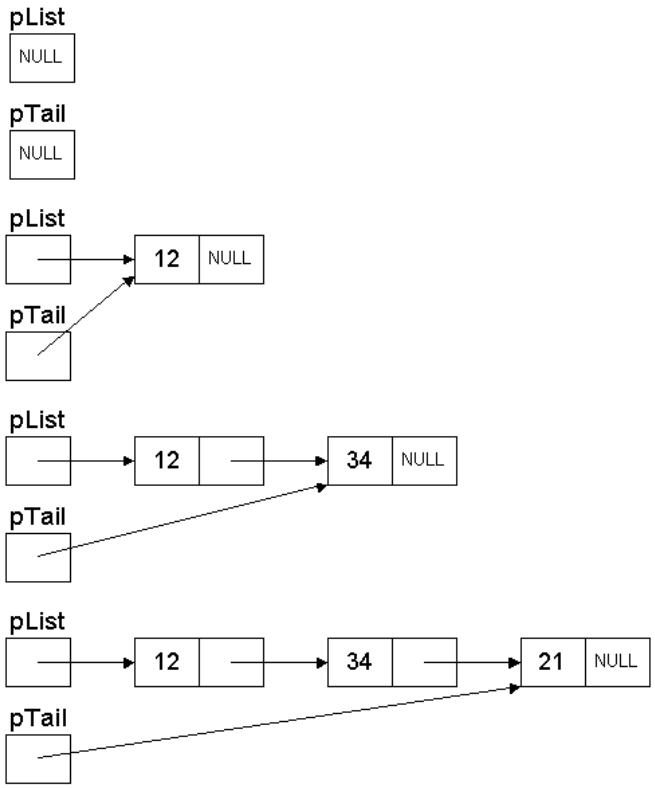
Let's address the last two points now. First, this one: ***"The time it takes to add a node to the end of the linked list takes longer as the list grows."***

This is simply because we are adding to the end and we don't have any immediate (random) access to the end. We only have immediate access to the first node; all of the other nodes must be accessed from the first one. If the list is long, this can take a while.

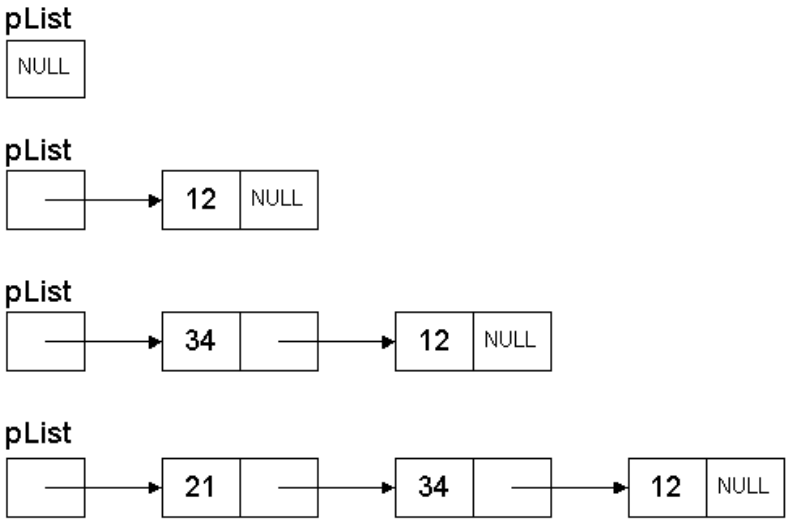
Solution #1: Maintain a pointer to the last node (tail).

We add a variable to track the tail:

```
Node *pList = 0; // empty list
Node *pTail = 0; // no tail yet
```



Solution #2: Insert at the head of the list instead of the tail. This is simpler yet. This has the "feature" that the items in the list will be *reversed.*



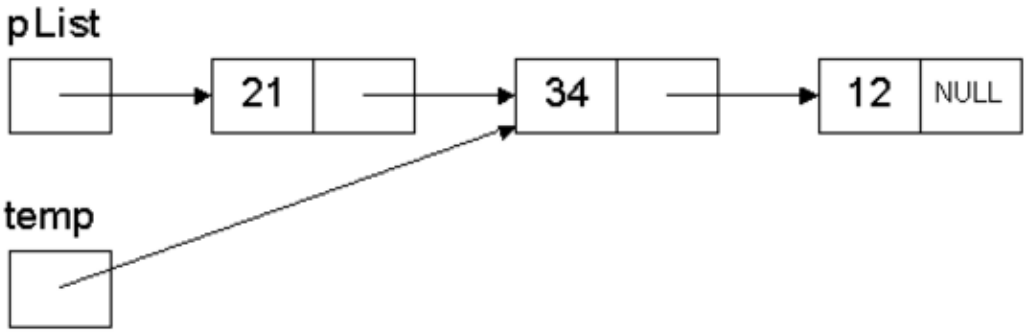
Freeing Nodes

Up until now, we haven't deleted any of the nodes. Since we called **`new`** for these nodes, we have to call **`delete`** when we're through. This is straight-forward using another while loop:

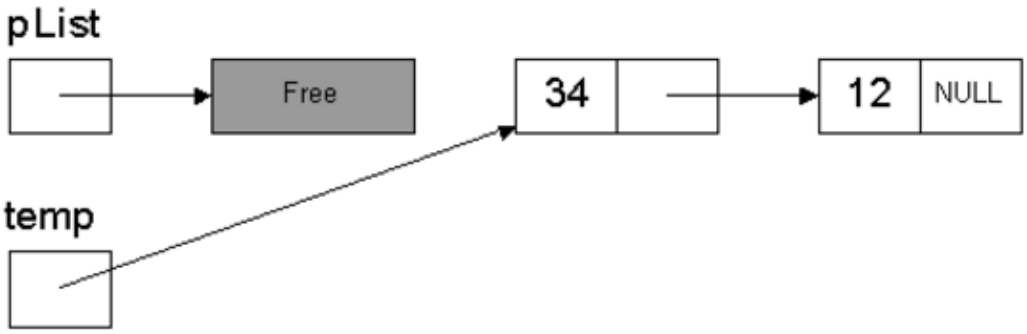
```
while (pList)
{
	Node *temp = pList->next;
	delete pList;
	pList = temp;
}
```

First Iteration (will delete the first node):

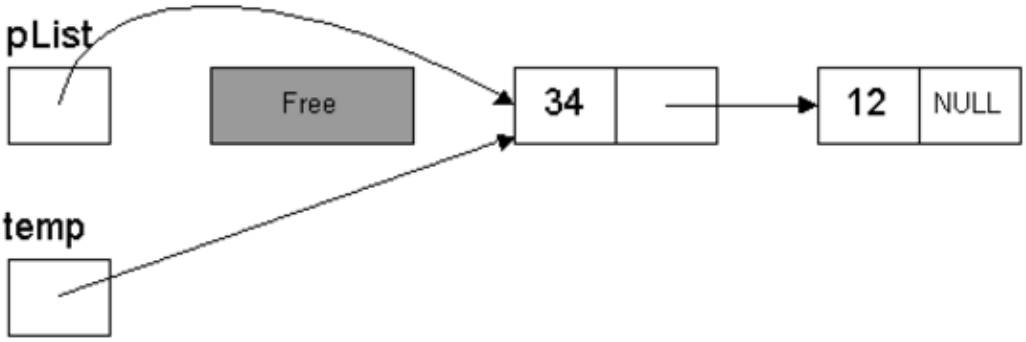
```
temp = pList->next;
```




```
delete pList;
```

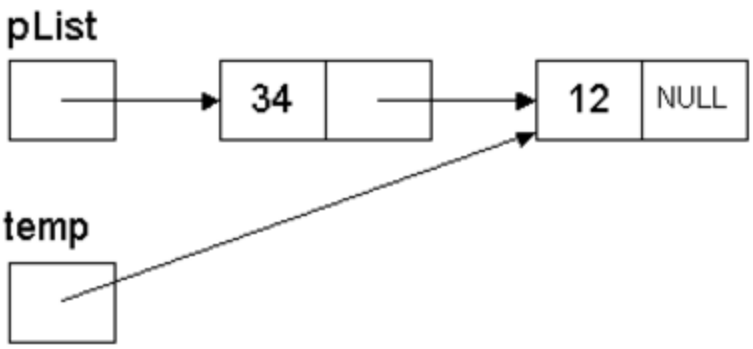



```
pList = temp;
```

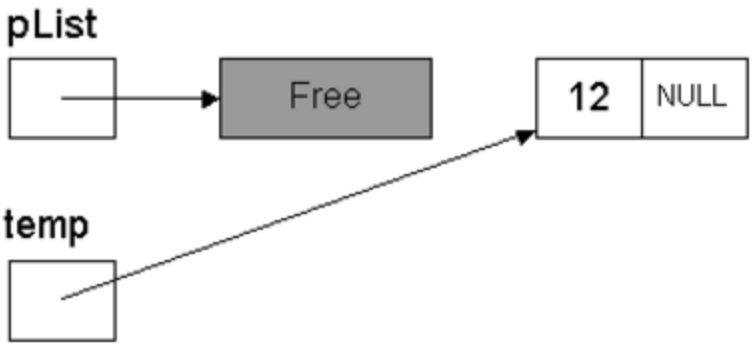


Second Iteration:

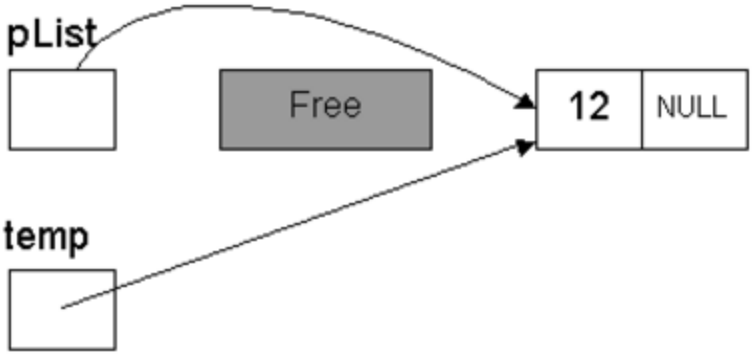
```
temp = pList->next;
```




```
delete pList;
```

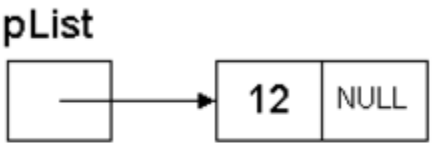


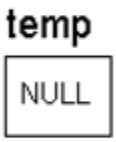

```
pList = temp;
```



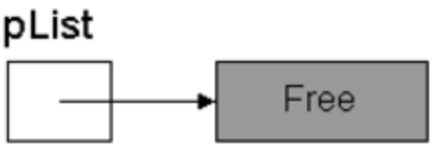
Third Iteration:

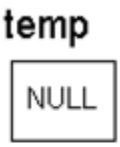
```
temp = pList->next;
```



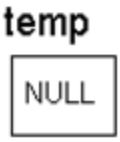



```
delete pList;
```






```
pList = temp;
```



End of while loop

Notes thus far:

- When we traverse (walk) a linked list, we always start from the beginning (head).
- We cannot "jump" to a particular node because we don't have random access (like arrays).
- Note that we cannot walk backwards through a singly linked list because we have no way to get from a node to the previous node.
- If you only need to move in one direction, a singly linked list might be enough.
- If you require bi-directional traversals, you will want to use a *doubly linked list*, which allows traversals in both directions.

An Ordered List

The previous examples have added the data (integers) to the list in the order they arrived from the user. (Inserting at the front of the list caused the data to be reversed.) This is no different than the way you would add elements to an array.

Suppose we want to keep the linked list sorted, from smallest to largest. This is the data that is entered by the user.

```
12 34 21 56 38 94 23 22 67 56 88 19 59 10 17
```

If you were to implement this with an array, describe the algorithm that you would use to keep the list sorted.

1. Ask the user if they want to input a number
2. While the user wants to input a number
    a. Read in a number
    b. ...
    c. ...
    etc...

Again, we just need to modify our **`while`** loop that adds a node to the list:

Code Change

```
// Allocate a new node struct (same for all nodes)
pNode = new Node;
pNode->number = number; // Initialize number
pNode->next = NULL;     // Initialize next

// If the list is empty, this is the first one
if (pList == NULL)
{
       pList = pNode;
}
else
{
       Node *temp = pList, *prev = NULL;

       // Find the correct position in the list for the new node.
       while (temp && (temp->number < pNode->number))
       {
          prev = temp;       // save previous (singly linked)
          temp = temp->next;
       }

       // If this number comes before the first one
       if (temp == pList)
       {
          pList = pNode;
       }
       else
       {
          prev->next = pNode; // Insert between current and prev node
       }

       pNode->next = temp;   // Next node is larger than new node
}
```

Output

```
10  12  17  19  21  22  23  34  38  56  56  59  67  88  94
```

If we insert a call to `print_list` after every insertion into the list, we can see the list evolve. The **bold** indicates newly inserted numbers:

```
pList → 12
pList → 12 → 34
pList → 12 → 21 → 34
pList → 12 → 21 → 34 → 56
pList → 12 → 21 → 34 → 38 → 56
pList → 12 → 21 → 34 → 38 → 56 → 94
pList → 12 → 21 → 23 → 34 → 38 → 56 → 94
pList → 12 → 21 → 22 → 23 → 34 → 38 → 56 → 94
pList → 12 → 21 → 22 → 23 → 34 → 38 → 56 → 67 → 94
pList → 12 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 67 → 94
pList → 12 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 67 → 88 → 94
pList → 12 → 19 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 67 → 88 → 94
pList → 12 → 19 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 59 → 67 → 88 → 94
pList → 10 → 12 → 19 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 59 → 67 → 88 → 94
pList → 10 → 12 → 17 → 19 → 21 → 22 → 23 → 34 → 38 → 56 → 56 → 59 → 67 → 88 → 94
```

Try doing *that* with an array! (By the way, how would you do this with an array?)

Creating Functions

The code that was shown thus far manipulates all of the nodes in the list directly. What this means is that the lists were not "passed" to another function to do the work. In real code, you would create functions to do things such as ***AddToEnd, AddToFront, DeleteFront, FindItem***, etc. The reason to do this is simple: we want to be able to re-use the functionality so that we can perform these operations on any list.

However, there is a slight caveat. Something that we've discussed many times before but still confuses many new programmers: Passing a pointer to a function. Remember these points about passing parameters to functions:

- If we want a function to change our data, we must pass a pointer to our data.
- Passing by value will cause the function to change the copy of the data.
- For example, if I want a function to change an integer, I would pass a pointer to the integer (e.g. the swap function).
- But, what if I want the function to change a pointer? If we just pass the pointer by value (a copy), then the function will change the copy, meaning the copy will point to something else, but the original pointer will be unchanged.
- If we want a function to change our pointer, we have to do the same thing: pass a pointer to the pointer.

With linked lists, we may want a function to add a new node to the front of a list. The way we do this is to point the new node's next pointer at the first node in the list, and then have the head pointer point at the new node. This is trivial and we saw it above in this page.

This means that, instead of passing the head node to the function, we need to pass a pointer to the head node to the function. This will allow the function to change what the head pointer will point at.

So, a function to add a node to the head of a list would be prototyped something like this (assuming a linked list of integers):

```
// Adds a node to the front of the list
void AddToFront(Node **ppList, Node *pNode);
```

1. **ppList** - a pointer to the head pointer (i.e. the address of the head pointer)
2. **pNode** - a pointer to the new node that will be inserted at the front of the list

So, a possible implementation of the function may look like this:

```
void AddToFront(Node **ppList, Node *pNode)
{
   // The new node's next pointer will point at the first node
   pNode->next = *ppList;

       // Now, the head pointer points at the new node, which is now at the front.
   // Notice that we are dereferencing the pointer to modify the original head pointer.
   // The client passed in the address of the head pointer so we could change it.
   *ppList = pNode;
}
```

As a rule of thumb, any function that might change what the head pointer is pointing at must take a pointer to a pointer to a node. If not, then the function will only change a copy of the head pointer, which, as everyone knows, will have no effect on the original head pointer. You would need to use a similar technique for a function that added to the end of a list as well. To help understand this concept, draw a diagram and see how passing pointers to pointers produces the desired results.

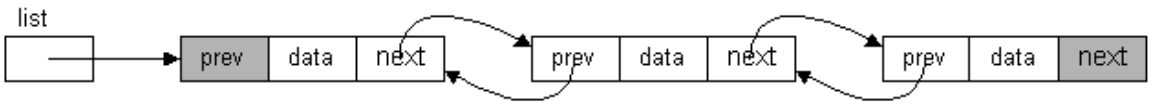
Note: There is nothing fancy or weird or exotic about passing pointers to pointers. The only way a function can change the original data is if it receives a pointer to the data. If the data happens to be a pointer itself, then a pointer to the pointer must be passed in order to change it. This is why some of the linked list functions must pass pointers to pointers to nodes.

Doubly Linked Lists

A *doubly linked list* is a list that has two pointers. In addition to the *next* pointer, it has a *previous* pointer. This allows you to traverse the list in both directions.

An example node structure for a doubly linked list:

```
struct Node
{
       int number;        // data portion
       struct Node *next; // node after this one
       struct Node *prev; // node before this one
};
```



Compared with singly linked lists, double linked lists:

- require an extra pointer for each node.
- require a more work at runtime to "hook-up" the extra pointer.
- are more flexible by allowing traversals in both directions.
- can be simpler to implement because nodes can access their next and previous node.

The implementations for functions to manipulate doubly linked lists would be similar to the singly linked lists functions, with the additional code for dealing with the previous pointer.

To summarize, linked lists:

- are an alternate data structure for storing related data.
- are more complex in structure and more complex in accessing than arrays.
- require the programmer to deal with the complexity. (They are not built into the language.)
- require more overhead than arrays.
- can grow and shrink in size at runtime.
- allow very efficient use of memory since you only allocate what you need (and delete when you're done.)
- are very efficient for inserting and deleting items at any point in the list.

Why not do away with arrays and use linked lists for all lists?