4. Data Structures

4.1. Structures

Structures are aggregate data types built using elements of primitive data types.

Structure are defined using the struct keyword:

E.g.
```
struct Time{
    int hour;
    int minute;
    int second;
};
```

The struct keyword creates a new user defined data type that is used to declare variables of an aggregate data type.

Structure variables are declared like variables of other types.

Syntax: struct <structure tag> <variable name>;

E.g.
```
struct Time timeObject,

struct Time *timeptr;
```

4.1.1. Accessing Members of Structure Variables

The Dot operator (.): to access data members of structure variables.

The Arrow operator (->): to access data members of pointer variables pointing to the structure.

E.g. Print member hour of timeObject and timeptr.

```
cout<< timeObject.hour; or

cout<<timeptr->hour;
```

TIP: timeptr->hour is the same as (*timeptr).hour.

The parentheses is required since (*) has lower precedence than (.).

4.1.2. Self-Referential Structures

Structures can hold pointers to instances of themselves.

```
struct list{
        char name[10];
        int count;
        struct list *next;
};
```

However, structures cannot contain instances of themselves.

4.2. Singly Linked Lists

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

Array vs. Linked lists

Arrays are *simple* and *fast* *but we m*ust specify their size at construction time. This has its own drawbacks. If you construct an array with space for *n*, tomorrow you may need *n*+1.Here comes a need for a more flexible system?

Advantages of Linked Lists

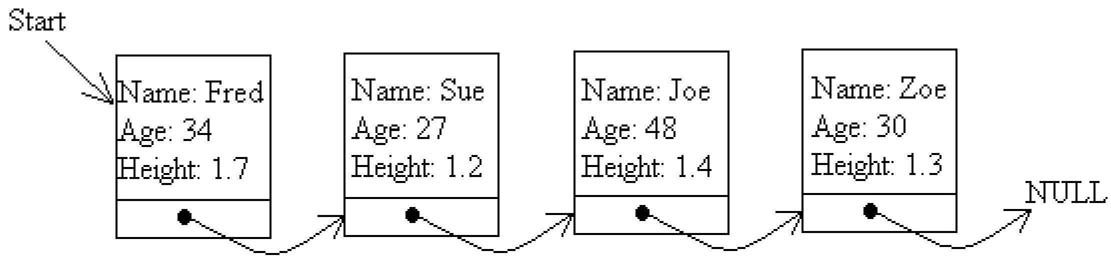
Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized.

A linked list is made up of a chain of nodes. Each node contains:

- the data item, and
- a pointer to the next node

4.2.1. Creating Linked Lists in C++

A linked list is a data structure that is built from structures and pointers. It forms a chain of "nodes" with pointers representing the links of the chain and holding the entire thing together. A linked list can be represented by a diagram like this one:



This linked list has four nodes in it, each with a link to the next node in the series. The last node has a link to the special value NULL, which any pointer (whatever its type) can point to, to show that it is the last link in the chain. There is also another special pointer, called Start (also called head), which points to the first link in the chain so that we can keep track of it.

4.2.2. Defining the data structure for a linked list

The key part of a linked list is a structure, which holds the data for each node (the name, address, age or whatever for the items in the list), and, most importantly, a pointer to the next node. Here we have given the structure of a typical node:

```
struct node
  {  char name[20];    // Name of up to 20 letters
     int age
     float height;     // In metres
     node *nxt;// Pointer to next node
  };
struct node *start_ptr = NULL;
```

The important part of the structure is the line before the closing curly brackets. This gives a pointer to the next node in the list. This is the only case in C++ where you are allowed to refer to a data type (in this case **node**) before you have even finished defining it!

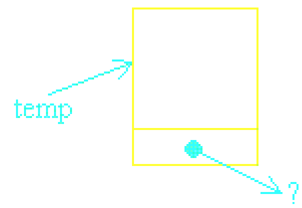
We have also declared a pointer called **start_ptr** that will permanently point to the start of the list. To start with, there are no nodes in the list, which is why **start_ptr** is set to NULL.

4.2.4. Adding a node to the list

The first problem that we face is how to add a node to the list. For simplicity's sake, we will assume that it has to be added to the end of the list, although it could be added anywhere in the list (a problem we will deal with later on).

Firstly, we declare the space for a pointer item and assign a temporary pointer to it. This is done using the **new** statement as follows:

temp = new node;



We can refer to the new node as ***temp**, i.e. **"the node that temp points to"**. When the fields of this structure are referred to, brackets can be put round the ***temp** part, as otherwise the compiler will think we are trying to refer to the fields of the pointer. Alternatively, we can use the arrow pointer notation.

That's what we shall do here.

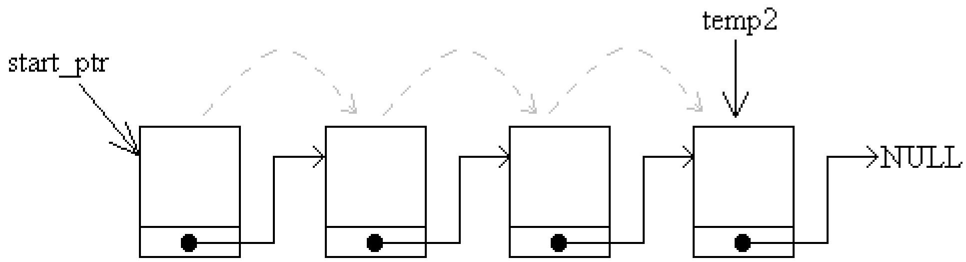
Having declared the node, we ask the user to fill in the details of the person, i.e. the name, age, address or whatever:

```
cout << "Please enter the name of the person: ";
cin >> temp->name;
cout << "Please enter the age of the person : ";
cin >> temp->age;
cout << "Please enter the height of the person : ";
cin >> temp->height;
temp->nxt = NULL;
```

The last line sets the pointer from this node to the next to NULL, indicating that this node, when it is inserted in the list, will be the last node. Having set up the information, we have to decide what to do with the pointers. Of course, if the list is empty to start with, there's no problem - just set the Start pointer to point to this node (i.e. set it to the same value as temp):

```
if (start_ptr == NULL)
   start_ptr = temp;
```

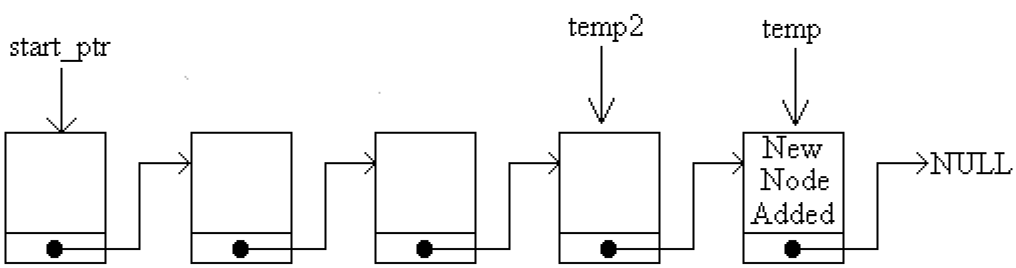
It is harder if there are already nodes in the list. In this case, the secret is to declare a second pointer, **temp2**, to step through the list until it finds the last node.




```
temp2 = start_ptr;
  // We know this is not NULL - list not empty!
while (temp2->nxt != NULL)
 {  temp2 = temp2->nxt;   // Move to next link in chain
 }
```

The loop will terminate when **temp2** points to the last node in the chain, and it knows when this happened because the **nxt** pointer in that node will point to NULL. When it has found it, it sets the pointer from that last node to point to the node we have just declared:

```
temp2->nxt = temp;
```



The link **temp2->nxt** in this diagram is the link joining the last two nodes. The full code for adding a node at the end of the list is shown below, in its own little function:

```
void add_node_at_end ()
 {  node *temp, *temp2;   // Temporary pointers

    // Reserve space for new node and fill it with data
    temp = new node;
    cout << "Please enter the name of the person: ";
    cin >> temp->name;
    cout << "Please enter the age of the person : ";
    cin >> temp->age;
    cout << "Please enter the height of the person : ";
    cin >> temp->height;
```

```
    temp->nxt = NULL;

    // Set up link to this node
    if (start_ptr == NULL)
       start_ptr = temp;
    else
      { temp2 = start_ptr;
        // We know this is not NULL - list not empty!
        while (temp2->nxt != NULL)
          {  temp2 = temp2->nxt;
             // Move to next link in chain
          }
        temp2->nxt = temp;
      }
  }
```

4.2.4. Displaying the list of nodes

Having added one or more nodes, we need to display the list of nodes on the screen. This is comparatively easy to do. Here is the method:

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the nxt pointer of the node it is currently indicating.
5. Jump back to step 2.

The temporary pointer moves along the list, displaying the details of the nodes it comes across. At each stage, it can get hold of the next node in the list by using the **nxt** pointer of the node it is currently pointing to. Here is the C++ code that does the job:

```
temp = start_ptr;
do
 {  if (temp == NULL)
      cout << "End of list" << endl;
    else
      {  // Display details for what temp points to
         cout << "Name : " << temp->name << endl;
         cout << "Age : " << temp->age << endl;
```

```
        cout << "Height : " << temp->height << endl;
        cout << endl;      // Blank line

        // Move to next node (if present)
        temp = temp->nxt;
      }
   } while (temp != NULL);
```

Check through this code, matching it to the method listed above. It helps if you draw a diagram on paper of a linked list and work through the code using the diagram.

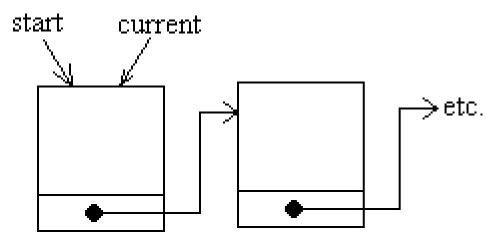
4.2.5. Navigating through the list

One thing you may need to do is to navigate through the list, with a pointer that moves backwards and forwards through the list, like an index pointer in an array. This is certainly necessary when you want to insert or delete a node from somewhere inside the list, as you will need to specify the position.

We will call the mobile pointer **current**. First of all, it is declared, and set to the same value as the **start_ptr** pointer:

```
node *current;
current = start_ptr;
```

Notice that you don't need to set current equal to the *address* of the start pointer, as they are both pointers. The statement above makes them both point to the same thing:



It's easy to get the current pointer to point to the next node in the list (i.e. move from left to right along the list). If you want to move current along one node, use the nxt field of the node that it is pointing to at the moment:

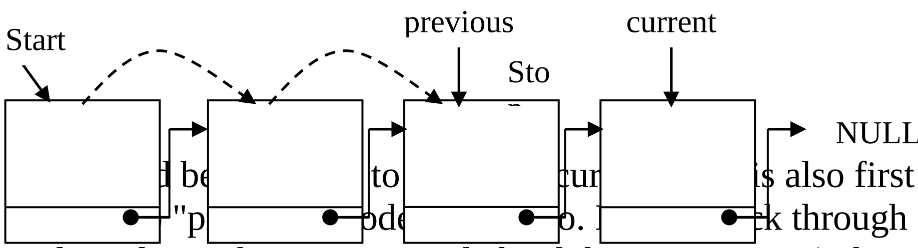
```
current = current->nxt;
```

In fact, we had better check that it isn't pointing to the last item in the list. If it is, then there is no next node to move to:

```
if (current->nxt == NULL)
```

```
        cout << "You are at the end of the list." << endl;
    else
        current = current->nxt;
```

Moving the current pointer back one step is a little harder. This is because we have no way of moving back a step automatically from the current node. The only way to find the node before the current one is to start at the beginning, work our way through and stop when we find the node before the one we are considering at the moment. We can tell when this happens, as the **nxt** pointer from that node will point to exactly the same place in memory as the current pointer (i.e. the current node).



First of d be to cur s also first the one. If it is, then "p ode o. k through all the nodes in turn until we detect that we are just behind the current one (Like a pantomime - "behind you!")

```
    if (current == start_ptr)
        cout << "You are at the start of the list" << endl;
    else
      { node *previous;     // Declare the pointer
        previous = start_ptr;

        while (previous->nxt != current)
          {    previous = previous->nxt;
          }
        current = previous;
      }
```

The else clause translates as follows: Declare a temporary pointer (for use in this else clause only). Set it equal to the start pointer. All the time that it is not pointing to the node before the current node, move it along the line. Once the previous node has been found, the current pointer is set to that node - i.e. it moves back along the list.

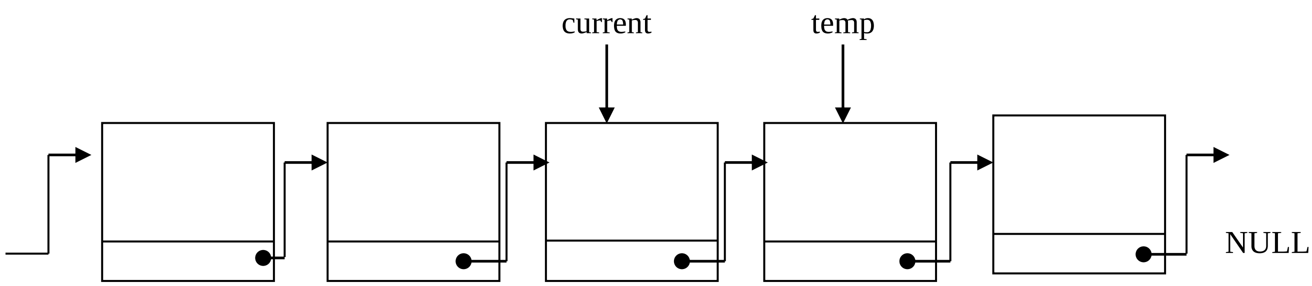
Now that you have the facility to move back and forth, you need to do something with it. Firstly, let's see if we can alter the details for that particular node in the list:

```
    cout << "Please enter the new name of the person: ";
```

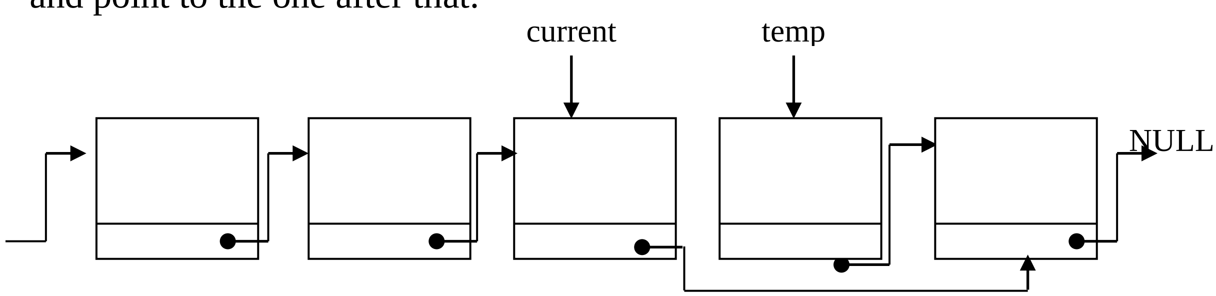
```
cin >> current->name;
cout << "Please enter the new age of the person : ";
cin >> current->age;
cout << "Please enter the new height of the person : ";
cin >> current->height;
```

The next easiest thing to do is to delete a node from the list directly after the current position. We have to use a temporary pointer to point to the node to be deleted. Once this node has been "anchored", the pointers to the remaining nodes can be readjusted before the node on death row is deleted. Here is the sequence of actions:

1. Firstly, the temporary pointer is assigned to the node after the current one. This is the node to be deleted:



2. Now the pointer from the current node is made to leap-frog the next node and point to the one after that:



3. The last step is to delete the node pointed to by **temp**.

Here is the code for deleting the node. It includes a test at the start to test whether the current node is the last one in the list:

```
if (current->nxt == NULL)
   cout << "There is no node after current" << endl;
else
```

```
    { node *temp;
      temp = current->nxt;
      current->nxt = temp->nxt;     // Could be NULL
      delete temp;
    }
```

Here is the code to ***add*** a node after the current one. This is done similarly, but we haven't illustrated it with diagrams:

```
  if (current->nxt == NULL)
      add_node_at_end();
  else
    { node *temp;
      new temp;
      get_details(temp);
      // Make the new node point to the same thing as
      // the current node
      temp->nxt = current->nxt;
      // Make the current node point to the new link
      // in the chain
      current->nxt = temp;
    }
```

We have assumed that the function **add_node_at_end()** is the routine for adding the node to the end of the list that we created near the top of this section. This routine is called if the current pointer is the last one in the list so the new one would be added on to the end.

Similarly, the routine **get_temp(temp)** is a routine that reads in the details for the new node similar to the one defined just above.

... and so ...

4.2.6. Deleting a node from the list

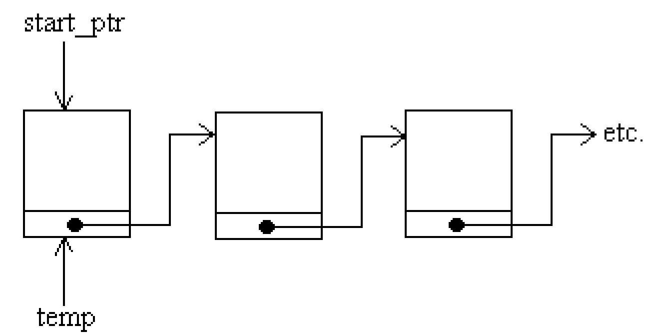
When it comes to deleting nodes, we have three choices: Delete a node from the start of the list, delete one from the end of the list, or delete one from somewhere in the middle. For simplicity, we shall just deal with deleting one from the start or from the end.

When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the **delete** instruction:

delete temp; // Release the memory pointed to by temp

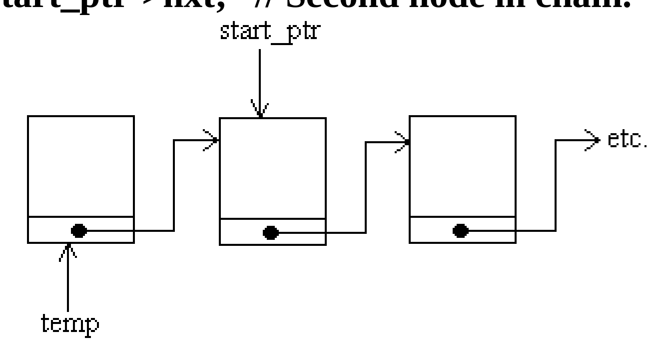
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. Here is how we go about deleting the first node in the linked list:

temp = start_ptr; // Make the temporary pointer
// identical to the start pointer

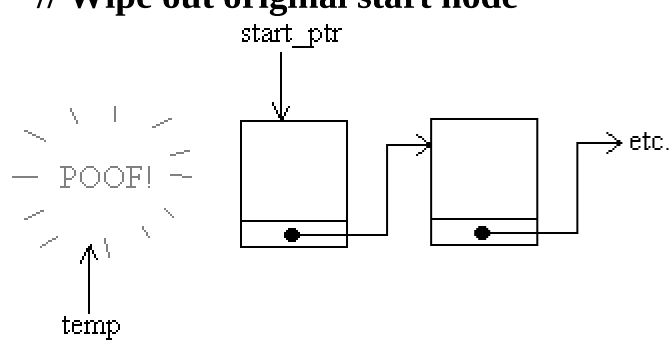


Now that the first node has been safely tagged (so that we can refer to it even when the start pointer has been reassigned), we can move the start pointer to the next node in the chain:

start_ptr = start_ptr->nxt; // Second node in chain.



delete temp; // Wipe out original start node



Here is the function that deletes a node from the start:

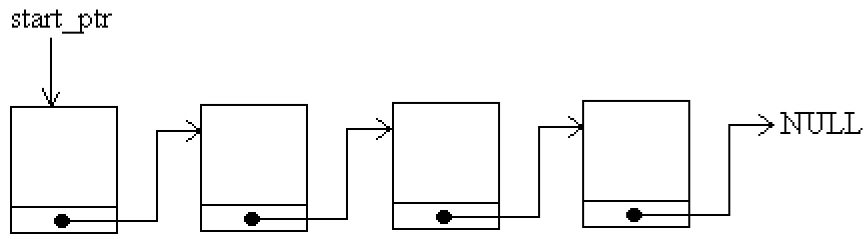
void delete_start_node()

```
{ node *temp;
  temp = start_ptr;
  start_ptr = start_ptr->nxt;
  delete temp;
}
```

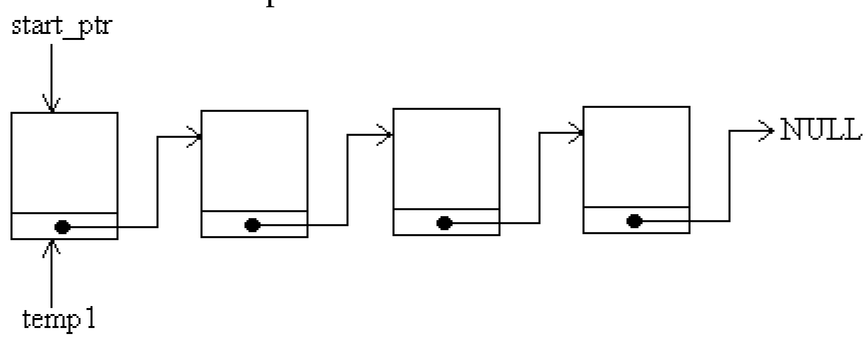
Deleting a node from the end of the list is harder, as the temporary pointer must find where the end of the list is by hopping along from the start. This is done using code that is almost identical to that used to insert a node at the end of the list. It is necessary to maintain two temporary pointers, **temp1** and **temp2**. The pointer **temp1** will point to the last node in the list and **temp2** will point to the previous node. We have to keep track of both as it is necessary to delete the last node and immediately afterwards, to set the **nxt** pointer of the previous node to NULL (it is now the new last node).

1. Look at the start pointer. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.
2. Make **temp1** point to whatever the start pointer is pointing to.
3. If the **nxt** pointer of what temp1 indicates is NULL, then we've found the last node of the list, so jump to step 7.
4. Make another pointer, **temp2**, point to the current node in the list.
5. Make **temp1** point to the next item in the list.
6. Go to step 4.
7. If you get this far, then the temporary pointer, **temp1**, should point to the last item in the list and the other temporary pointer, **temp2**, should point to the last-but-one item.
8. Delete the node pointed to by **temp1**.
9. Mark the **nxt** pointer of the node pointed to by **temp2** as NULL - it is the new last node.

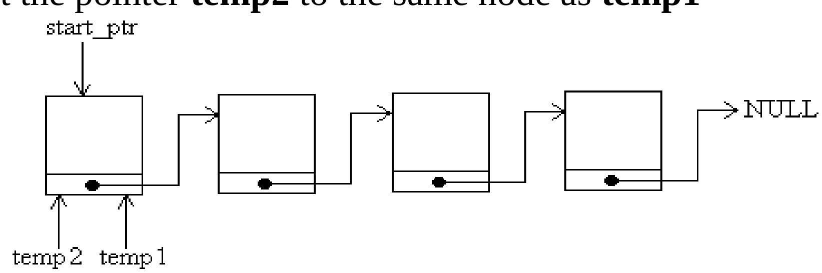
Let's try it with a rough drawing. This is always a good idea when you are trying to understand an abstract data type. Suppose we want to delete the last node from this list:



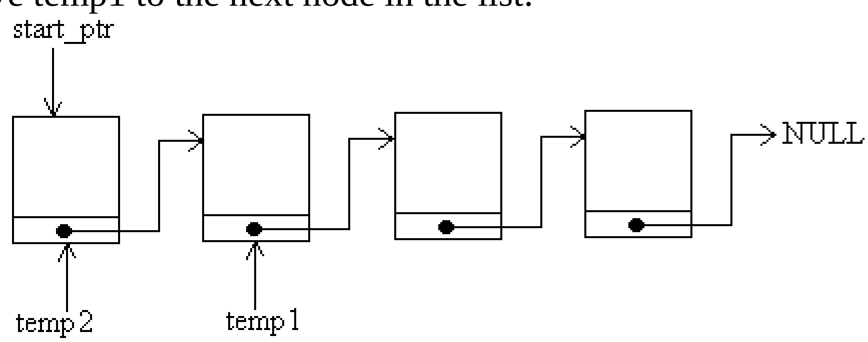
Firstly, the start pointer doesn't point to NULL, so we don't have to display a "Empty list, wise guy!" message. Let's get straight on with step2 - set the pointer **temp1** to the same as the start pointer:



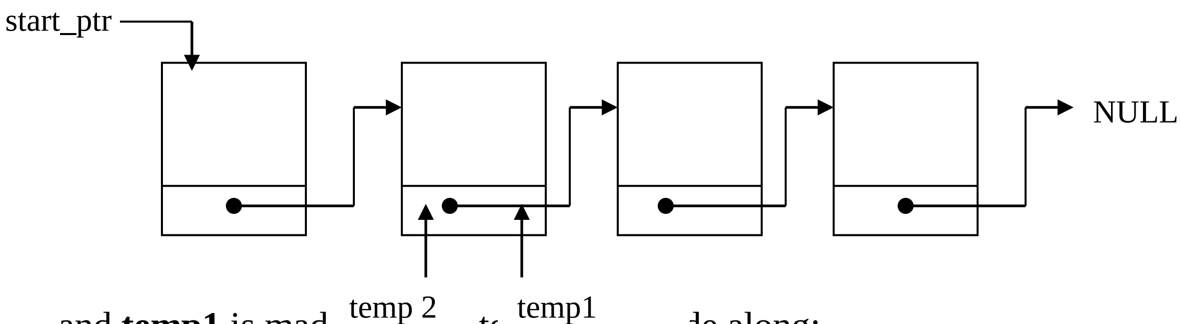
The **nxt** pointer from this node isn't NULL, so we haven't found the end node. Instead, we set the pointer **temp2** to the same node as **temp1**



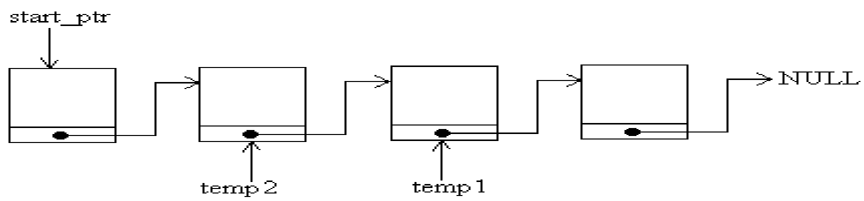
and then move temp1 to the next node in the list:



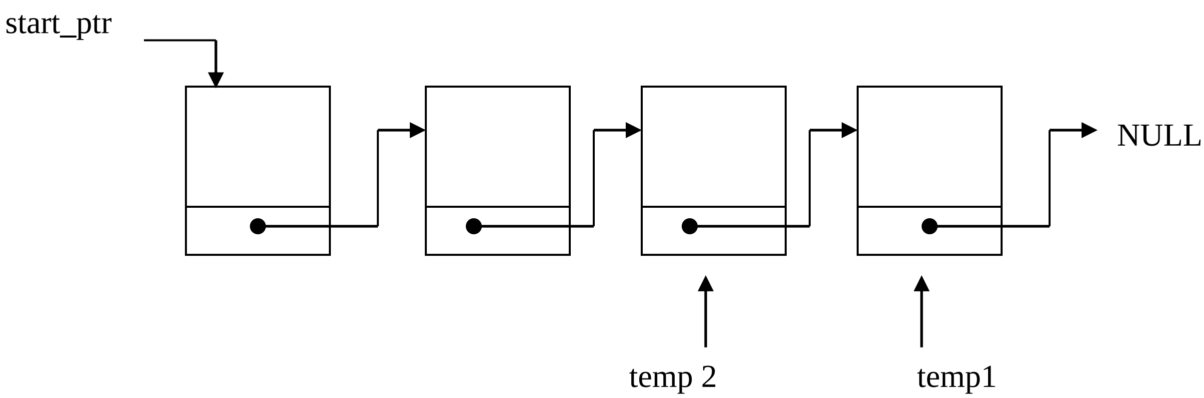
Going back to step 3, we see that temp1 still doesn't point to the last node in the list, so we make temp2 point to what temp1 points to



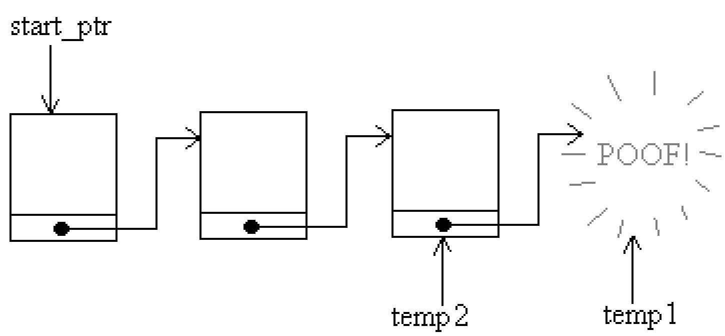
and **temp1** is mad tc le along:



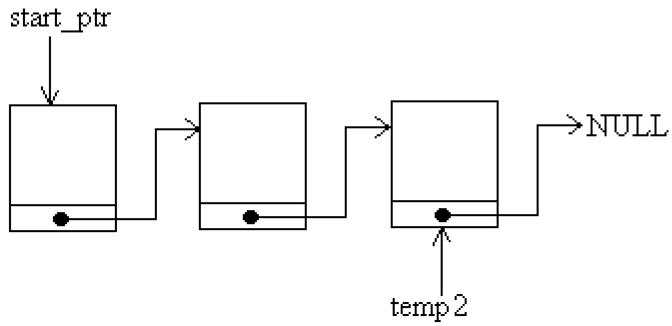
Eventually, this goes on until **temp1** really is pointing to the last node in the list, with **temp2** pointing to the penultimate node:



Now we have reached step 8. The next thing to do is to delete the node pointed to by **temp1**



and set the **nxt** pointer of what **temp2** indicates to NULL:



We suppose you want some code for all that! All right then ....

```
void delete_end_node()
  { node *temp1, *temp2;
   if (start_ptr == NULL)
       cout << "The list is empty!" << endl;
   else
     { temp1 = start_ptr;
       while (temp1->nxt != NULL)
         { temp2 = temp1;
           temp1 = temp1->nxt;
         }
       delete temp1;
       temp2->nxt = NULL;
     }
  }
```

The code seems a lot shorter than the explanation!

Now, the sharp-witted amongst you will have spotted a problem. If the list only contains one node, the code above will malfunction. This is because the function goes as far as the **temp1 = start_ptr** statement, but never gets as far as setting up **temp2**. The code above has to be adapted so that if the first node is also the last (has a NULL **nxt** pointer), then it is deleted and the **start_ptr** pointer is assigned to NULL. In this case, there is no need for the pointer **temp2**:

```
void delete_end_node()
  { node *temp1, *temp2;
   if (start_ptr == NULL)
       cout << "The list is empty!" << endl;
   else
     { temp1 = start_ptr;
```

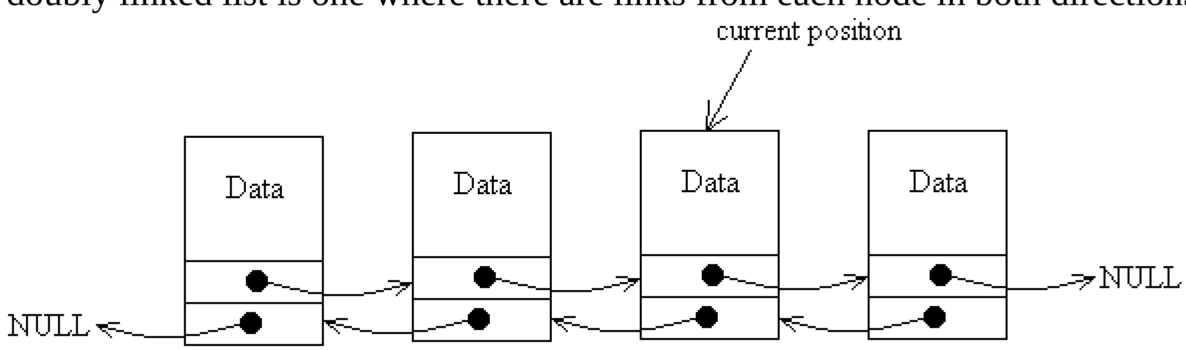
```
if (temp1->nxt == NULL)     // This part is new!
  { delete temp1;
    start_ptr = NULL;
  }
else
{ while (temp1->nxt != NULL)
      { temp2 = temp1;
        temp1 = temp1->nxt;
      }
    delete temp1;
    temp2->nxt = NULL;
  }
}

}
```

4.4. Doubly Linked Lists

That sounds even harder than a linked list! Well, if you've mastered how to do singly linked lists, then it shouldn't be much of a leap to doubly linked lists

A doubly linked list is one where there are links from each node in both directions:



You will notice that each node in the list has two pointers, one to the next node and one to the previous one - again, the ends of the list are defined by NULL pointers. Also there is no pointer to the start of the list. Instead, there is simply a pointer to some position in the list that can be moved left or right.

The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed.

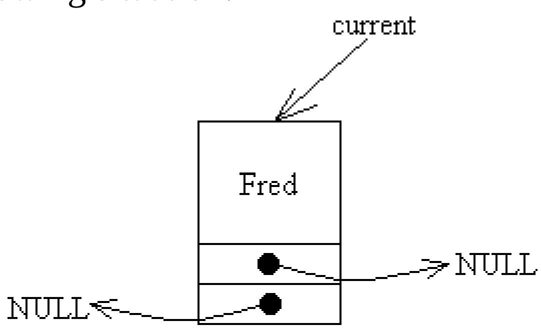
With the doubly linked list, we can move the current pointer backwards and forwards at will.

4.4.1. Creating Doubly Linked Lists

The nodes for a doubly linked list would be defined as follows:

```
struct node{
  char name[20];
  node *nxt;    // Pointer to next node
  node *prv;    // Pointer to previous node
 };
node *current;
current = new node;
current->name = "Fred";
current->nxt = NULL;
current->prv = NULL;
```

We have also included some code to declare the first node and set its pointers to NULL. It gives the following situation:



We still need to consider the directions 'forward' and 'backward', so in this case, we will need to define functions to add a node to the start of the list (left-most position) and the end of the list (right-most position).

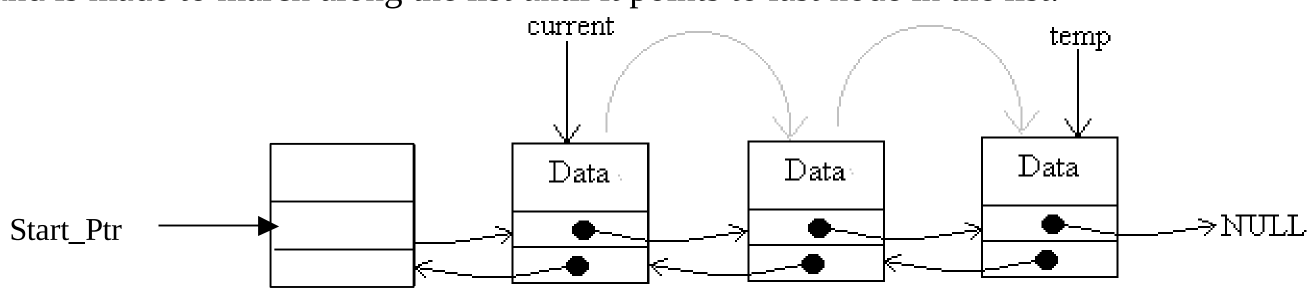
4.4.2. Adding a Node to a Doubly Linked List

```
void add_node_at_start (string new_name)
 { // Declare a temporary pointer and move it to the start
  node *temp = current;
  while (temp->prv != NULL)
   temp = temp->prv;
  // Declare a new node and link it in
  node *temp2;
  temp2 = new node;
  temp2->name = new_name;  // Store the new name in the node
```

```
  temp2->prv = NULL;      // This is the new start of the list
  temp2->nxt = temp;      // Links to current list
  temp->prv = temp2;
 }


void add_node_at_end ()
 { // Declare a temporary pointer and move it to the end
  node *temp = current;
  while (temp->nxt != NULL)
   temp = temp->nxt;
  // Declare a new node and link it in
  node *temp2;
  temp2 = new node;
  temp2->name = new_name;  // Store the new name in the node
  temp2->nxt = NULL;      // This is the new start of the list
  temp2->prv = temp;      // Links to current list
  temp->nxt = temp2;
 }
```

Here, the new name is passed to the appropriate function as a parameter. We'll go through the function for adding a node to the right-most end of the list. The method is similar for adding a node at the other end. Firstly, a temporary pointer is set up and is made to march along the list until it points to last node in the list.



After that, a new node is declared, and the name is copied into it. The nxt pointer of this new node is set to NULL to indicate that this node will be the new end of the list.
The prv pointer of the new node is linked into the last node of the existing list.
The nxt pointer of the current end of the list is set to the new node.